

THESE

présentée à

L' UNIVERSITE PIERRE ET MARIE CURIE (PARIS VI)

pour l'obtention du

DOCTORAT TROISIEME CYCLE

Spécialité : MATHEMATIQUES

Mention : Informatique

par M. GAUTHIER Michel

Sujet de la thèse : Quelques outils et méthodes
pour programmer avec des pointeurs

Soutenue le 19/10/1981 devant la Commission

composee de: Président M. J. ARSAC

Examineurs: MM. J.C. BOUSSARD
B. ROBINET

THESE

présentée à

L'UNIVERSITE PIERRE ET MARIE CURIE (PARIS VI)

pour l'obtention du

DOCTORAT TROISIEME CYCLE

Spécialité: MATHEMATIQUES

Mention: Informatique

par M. GAUTHIER Michel

Sujet de la thèse: Quelques outils et méthodes
pour programmer avec des pointeurs

Soutenue le 19/10/1981 devant la Commission

composée de: Président M. J. ARSAC

Examineurs: MM. J.C. BOUSSARD
B. ROBINET

Je remercie Monsieur le Professeur J. ARSAC de me faire l'honneur de présider le jury de cette thèse. Qu'il trouve ici l'expression de mon admiration pour l'ouverture dont il fait preuve à l'égard de tous les domaines de l'informatique.

Que Monsieur le Professeur B. ROBINET trouve ici le témoignage de ma reconnaissance pour l'acceptation et le suivi de mes recherches.

Monsieur le Professeur J.-C. BOUSSARD a bien voulu accepter de s'intéresser à mes travaux et de participer au jury. Qu'il en soit remercié.

Que l'on me permette une pensée particulière pour ma famille et mes amis qui ont su me supporter avec patience pendant la gestation de ce travail.

Ce travail n'aurait pu être mené à bien sans l'aide du personnel du Service des Moyens de Calcul qui a toujours accepté aimablement l'utilisateur exigeant que je suis, ni de celui de l'imprimerie dont la devise semble être de toujours faire bonne impression. Merci à tous.

A mon père

A tous ceux qui ont suivi
avec lui
le chemin

D' Eysses
de Compiègne
de Dachau.



INTRODUCTION

1.1. LES METHODES DE PROGRAMMATION SURE.	1
1.1.1. Specifier et formaliser.	1
1.1.2. Verifier, prouver.	2
1.1.3. Limites concretes.	2
1.1.4. Construire et prouver.	2
1.1.5. Transformer.	3
1.1.6. Synthese.	3
1.2. L'ABSTRACTION EN PROGRAMMATION.	4
1.2.1. Les structures de controle.	4
1.2.2. Les types abstraits.	4
1.3. POURQUOI PARLER DES POINTEURS ?	5
1.3.1. Qu'est-ce qu'un pointeur ?	5
1.3.2. La place des pointeurs dans la programmation.	7
1.3.3. Comment programme-t-on avec des pointeurs ?	8
1.4. PRESENTATION D'UN EXEMPLE	10
1.4.1. Enonce et formalisation.	10
1.4.1.1. Analyse du probleme.	10
1.4.1.2. Synthese.	11
1.4.1.3. Premiere construction de programme.	11
1.4.2. Resolution "informatique" du probleme.	12
1.4.3. Problemes de types et de representation.	13
1.4.4. Consequences de l'etude rapide de cet exemple.	15
1.5. PLAN COMMENTE.	15

ADAPTATION DU CALCUL DES PREDICATS A LA FORMALISATION DES POINTEURS

2.1. IDEE DIRECTRICE	1
2.2. CALCUL DES PREDICATS AVEC TYPES D'OBJETS	1
2.2.1. Definitions generales.	1
2.2.2. Definition d'un calcul type dans un calcul non type.	3
2.2.3. Introduction de fonctions dans le calcul type.	5

2.3. CALCUL DES PREDICATS POUR LES STRUCTURES CHAINEES	6
2.3.1. Elements de base	6
2.3.2. Langage de formules	8
2.3.3. Axiomatique minimale	9
2.3.4. Realisations	9
2.3.5. Resume de la construction	10
2.4. UTILISATION ET COMMENTAIRES	11
2.4.1. Completude, consistance, independance d'axiomes	11
2.4.2. Un exemple d'utilisation en calcul etendu	12
2.4.3. Introduction de proprietes du deuxieme ordre.	13
2.4.4. Le meme exemple en calcul strict.	14
2.4.5. Modeles infinis.	15
2.5. RESUME ET CONCLUSION DU CHAPITRE	15

STRUCTURES CHAINEES

3.1. IDEE DIRECTRICE.	1
3.2. DEFINITION DES STRUCTURES CHAINEES.	1
3.2.1. Squelette.	1
3.2.2. Constructions	3
3.2.3. Axiomatique	4
3.2.4. References de repere, variables de repere	5
3.2.5. Operateurs sur les constructions	6
3.2.5.1. Operateur CREER(r)	7
3.2.5.2. Operateur DETRUIRE(r)	8
3.2.5.3. Operateur VALEUR(x,v)	9
3.2.5.4. Operateur MODIFIER(r,x)	9
3.2.5.5. Sous-programmes.	10
3.3. DEFINITION EN STRUCTURE CHAINEE DE L'EXEMPLE DE 1.4	10
3.3.1. Definition de la structure.	11
3.3.2. Raffinement de l'algorithme	12
3.3.3. Demonstration de cet algorithme	14

3.4. COMPLEMENTS DIVERS.	15
3.4.1. Problemes de destruction - jonctions "flottantes"	15
3.4.1.1. Divers choix possibles de destruction.	15
3.4.1.2. Notion de reference flottante.	16
3.4.1.3. Mise en oeuvre concrete.	17
3.4.2. Application : detruire les tables d'un compilateur.	18
3.4.3. Operateurs de modification des structures.	19
3.4.3.1. Ajouter un type, ajouter un moule.	19
3.4.3.2. Suppression d'un moule de variable	19
3.4.3.3. Suppression d'un moule propre	20
3.4.3.4. Suppression d'un type	20

REALISATION DES STRUCTURES CHAINEES

4.1. CADRE DU TRAVAIL	1
4.1.1. STRUCTURE DE BASE	2
4.1.2. Dans des langages avec pointeurs.	3
4.1.3. Representation avec des tableaux.	4
4.1.4. Realisation sur disque.	5
4.1.4.1. Representation non optimisee.	6
4.1.4.2. Representation totalement optimisee.	8
4.1.4.3. Autres choix de representations.	10
4.1.4.4. Conclusion sur les representations permanentes.	10
4.2. OPERATEURS SUR LA STRUCTURE	11
4.2.1. Addition et suppression de moules.	11
4.2.2. Addition et suppression de moules de variables.	12
4.2.3. Ajout et suppression de types.	13
4.3. REMARQUES GENERALES SUR LES REPRESENTATIONS	14

STRUCTURES DE BASE CLASSIQUES

5.1. LISTE LINEAIRE: DEFINITION AXIOMATIQUE	1
5.1.1. Systeme formel de base.	1
5.1.2. Forme normale.	2
5.1.3. Liste lineaire ordonnee.	4
5.1.4. Forme normale des listes ordonnees.	4
5.1.5. Proprietes complementaires des listes lineaires ordonnees.	6
5.1.6. Independance des axiomes.	7
5.2. LISTE LINEAIRE : DEFINITION FONCTIONNELLE	8
5.2.1. Definitions generales et exemples.	8
5.2.2. Definition comme structure chainee.	9
5.2.3. Principales proprietes.	10
5.2.4. Liste lineaire double.	11
5.2.5. Construction de listes lineaires chainees.	12
5.2.5.1. "ajouter X derriere Y defini"	12
5.2.5.2. "ajouter X en tete"	14
5.2.5.3. "retirer derriere Y defini"	14
5.2.5.4. "retirer en tete"	15
5.2.6. Listes lineaires ordonnees.	15
5.3. LISTES BINAIRES (et K-aires)	16
5.3.1. Definition des listes K-aires.	16
5.3.2. Liste binaire ordonnee.	18
5.3.3. Cas particulier ou $K=1$	18
5.3.4. Exemple concret de K-arithme non binaire.	18
5.3.5. Proprietes et utilisation des listes K-aires.	20
5.3.6. Exemple de demonstration d'algorithme sur les listes.	21

ADAPTATION DU LANGAGE PASCAL AUX STRUCTURES CHAINEES

6.1. GENERALITES	1
6.2. REDEFINITION DE LA COMPATIBILITE DES TYPES	1
6.2.1. Situation existante.	1
6.2.2. Arborecences de types.	2
6.2.3. Restriction des operateurs disponibles.	3
6.2.4. Combinaison des deux propositions.	5

6.3.	GESTION FINE DES VARIABLES	5
6.3.1.	Situation actuelle.	5
6.3.2.	Differents statuts de variables.	5
6.3.3.	Instructions declaratives.	6
6.3.4.	Balayage explicite des listes.	7
6.4.	GESTION DES MOULES PROPRES	9
6.4.1.	Principe general.	9
6.4.2.	Mise en oeuvre de l'addition de moules.	9
6.4.3.	Compilation des moules temporaires.	10
6.4.4.	Declaration des globaux.	11
6.4.5.	Moules comme arguments.	12
6.4.6.	Remarque sur la declaration des files.	13
6.4.7.	Un exemple d'utilisation.	13
6.5.	ADDITIONS DIVERSES	14
6.5.1.	Verification des dereperages.	14
6.5.2.	Ajout et retrait de types.	14

TRAITEMENT COMPLET D'UN EXEMPLE

7.1.	ENONCE ET FORMALISATION	1
7.1.1.	Expose informel de la situation.	1
7.1.2.	Formalisation du probleme.	2
7.1.3.	Enumeration des circuits.	3
7.2.	ALGORITHME EXPRIME EN STRUCTURES CHAINEES	5
7.2.1.	Elaboration des representations.	5
7.2.2.	Quelques extraits de l'algorithme en listes.	6
7.2.3.	Definition de la structure chaine.	6
7.2.4.	Axiomatique.	10
7.2.5.	Destruction de la structure.	12
7.3.	PROGRAMME, RESULTATS ET ENSEIGNEMENTS	12

ANNEXE : EXTRAITS DE PROGRAMME

INTRODUCTION

Ce memoire est consacre a la programmation, c'est-a-dire a l'ensemble de ce processus partant d'un probleme concret et aboutissant a un programme realisant une fonction associee a ce probleme. Plus precisement, nous presentons des methodes et des outils visant a ameliorer les rapports entre les formalisations de l'objet concret et leurs representations dans les langages de programmation.

1.1. LES METHODES DE PROGRAMMATION SURE.

Pour etre utile un programme doit realiser effectivement la fonction desiree. C'est une verite premiere mais il est toujours necessaire de la rappeler. De nombreux travaux ont ete consacres aux moyens de satisfaire cette contrainte fondamentale.

1.1.1. Specifier et formaliser.

Pour etre transformable en un programme un probleme doit d'abord etre formalise, c'est-a-dire decrit dans un langage adapte qui pourra souvent etre celui d'une theorie mathematique connue. Pour la classe de problemes qui nous interessera, ce langage pourra etre celui de la theorie des ensembles et de l'arithmetique des fonctions calculables.

En raison des ambiguïtes de formalisation du probleme concret, il est souvent necessaire d'introduire une phase preliminaire de specification formelle decrivant le probleme en termes non ambigus. Divers langages ont ete definis pour cela [WAR77, LAU76, ABR77].

Notre travail n'abordera cet aspect que pour illustrer la necessite de minimalite de la formalisation: elle doit exprimer tout le probleme, mais sans information inutile sur les phases ulterieures de la programmation.

1.1.2. Verifier, prouver.

Pour justifier qu'un programme realise la fonction desiree il est necessaire, a l'autre bout du processus, de pouvoir associer a tout programme une signification dans un cadre formel, avec un ensemble de regles de preuve, des systemes automatiques de verification. Une retombee de ces travaux dans le domaine concret consiste a definir des techniques de synthese ou de transformation de programme.

Trois grandes voies ont ete explorees dans ce but: - la methode des assertions [FL067] aboutissant a la definition axiomatique des langages [HOA69]; - la technique du "point fixe" [SC071]; - les arbres de programme [COU77].

Dans le cadre precis de notre travail, c'est surtout la definition axiomatique des langages qui a ete abordee [HOA73, DEM77], ainsi qu'une voie recursive [WIR76] aboutissant a l'emploi du "point fixe".

1.1.3. Limites concretes.

Ces diverses approches ont toutes montre leur interet theorique et leur capacite a resoudre de petits problemes. Elles echouent pratiquement des que la complexite de l'algorithme ou des donnees augmente.

En particulier les definitions semantiques des langages ne permettent generalement pas de construire un programme pour resoudre un vrai probleme.

Il devient donc necessaire de decomposer le probleme a resoudre en sous-problemes plus simples, jusqu'a obtenir soit des situations deja connues et verifiees, soit des cas suffisamment simples pour etre demontres directement.

1.1.4. Construire et prouver.

Necessite et possibilite de decomposer le probleme s'expriment dans la methode appelee "programmation structuree" [DAH72] et que nous resumerons en trois idees-forces: - on construit les programmes par une demarche descendante en decomposant a chaque fois en sous-problemes plus simples ou resolus; - on conserve l'indépendance des sous-problemes chaque fois qu'elle existe; - on peut alors prouver par toute methode appropriee le passage ("raffinement") d'un programme au suivant.

La puissance concrete de ces principes est de permettre dans la pratique la construction et simultanement la preuve a un niveau convaincant [COR78] de programmes de complexite quelconque. Dans cette pratique il n'est pas question de fournir une preuve formelle irrefutable mais, conformement a la pratique mathematique courante, de fournir une preuve reconnue par tous ses utilisateurs, sans le recours permanent a la preuve formelle. Chaque pas de ce processus peut utiliser pour ses besoins propres et independamment des autres pas l'une ou l'autre des voies habituelles de demonstration ou construction.

1.1.5. Transformer.

A toute etape de construction de programmes, il est possible d'employer des transformations qui ne modifient pas la fonction realisee par le programme, mais seulement son ecriture [BUR76, ARS78]. Les transformations portant sur la syntaxe sont universellement applicables aux diverses classes d'algorithmes, tandis que les transformations semantiques utilisent les proprietes des sous-algorithmes (absence de modifications de variables, permutabilite).

1.1.6. Synthese.

Notre but n'est pas ici de comparer ni de choisir entre ces diverses conceptions et leurs diverses constructions theoriques fondamentales. Dans la realite quotidienne de l'ecriture de programmes utiles, les unes et les autres peuvent trouver leur propre champ d'application.

C'est en ce sens que nous situerons les travaux exposes ici dans le cadre general de la programmation structuree, schema global du processus, les autres methodes fournissant les outils de celui-ci. En particulier, traitant des pointeurs, nous leur associerons une formalisation a un certain niveau de la demarche descendante, entre les types abstraits que nous allons evoker immediatement et les langages de programmation.

1.2. L'ABSTRACTION EN PROGRAMMATION.

Le processus de programmation a aussi largement beneficie du mecanisme d'abstraction, qui permet de deccrire de maniere synthetique les proprietes des objets d'une certaine classe, independamment des details qui les distinguent. Ce mecanisme d'abstraction, commun a toutes les sciences, s'applique en informatique separement aux programmes et aux donnees, ou il conduit respectivement aux notions de structure de controle et de type abstrait.

1.2.1. Les structures de controle.

La remarque fondamentale de la structuration des programmes est qu'on peut exprimer tout algorithme sequentiel en combinant des operations elementaires a l'aide des trois structures de concatenation (";") de selection (binaire "if" ou multiple "case") et d'iteration ("while") [BOH66].

Cette remarque est a la base de l'extension du langage Pascal [WIR73, HOA73] notamment comme base presque universelle de tout nouveau langage. Mais ce succes n'empeche pas le maintien en Pascal de l'ancienne structure de saut explicite ("goto") et son emploi effectif.

La raison de ce maintien du saut effectif est a trouver a notre sens dans la lourdeur des trois structures de base pour des situations qui s'accomoderaient mieux de la boucle avec sortie a plusieurs niveaux. On pourrait tout aussi bien definir une structure de controle adaptee a la recuperation apres erreur et demontrer l'adequation du saut pour la compiler.

En definitive il n'y a pas de raison fondamentale lorsque le probleme le justifie pour ne pas accepter de construire de nouvelles structures de controle avec les moyens de leur realisation. Nous allons retrouver un cas analogue avec la structuration des donnees.

1.2.2. Les types abstraits.

Les objets manipules par les programmes ont ete eux aussi soumis au mecanisme d'abstraction, qui a souligne l'importance des operateurs de base et de leurs proprietes, independamment de leur representation et de leur nature.

Ce mecanisme apparu dans les langages Simula-67 et Algol-68 a abouti au concept de type abstrait [LIS75, WUL75, GUT75]. De nombreux langages ont ete construits ou cette notion tient une grande place: Clu, Alphard, Soc, Gypsy et tout recemment Ada.

Pourtant, a la difference des structures de controle, les types abstraits n'ont pas remplace les anciennes methodes. Leur utilisation est reelle dans le processus de raffinement du programme a la main, mais la construction d'un langage industriel ou ils remplaceraient fondamentalement les anciennes structures reste a faire, meme apres Ada. A notre sens des difficultes demeurent dans la construction de structures complexes ou interviennent des objets organises dans plusieurs types, ainsi que le caractere dynamique de certains types (voir notre exemple de 1.4). Il est donc utile de savoir realiser de tels types ou d'en definir de nouveaux avec les moyens de leurs realisation, comme pour les structures de donnees.

La notion de pointeur, qui est pour nous l'homologue pour les donnees de la rupture de sequence des programmes, demeure necessaire pour la realisation (automatique ou a la main) des types abstraits et la preuve de correction de celle-ci, ainsi que comme fondement d'eventuelles directives de compilation. Au lieu de les definir au niveau de la semantique des langages [HOA69, DEM77] ou par une voie par une voie recursive [WIR76] nous les introduirons comme un nouveau type abstrait de nature particuliere, ce qui les rendra mieux aptes a ces taches.

1.3. POURQUOI PARLER DES POINTEURS ?

1.3.1. Qu'est-ce qu'un pointeur ?

Nous nous interessons ici a une classe de problemes et a leur resolution sur une classe de machines. Le terme utilise en general pour ces problemes est le qualificatif "non numerique". La theorie mathematique sous-jacente est la theorie des ensembles. Parmi eux on rencontre classiquement les diverses sortes de graphes et une partie des representations pour les structures de donnees de base. Cependant nous accorderons plus d'attention aux structures complexes, ou plusieurs ensembles doivent etre structures de plusieurs manieres.

Les machines utilisees se caracterisent par le caractere de l'acces a la memoire contenant les donnees. Cette memoire peut etre definie (parfois de plusieurs manieres) comme un tableau lineaire d'elements de base. On accede ainsi a chaque element de base par l'intermediaire de son "adresse" et l'ensemble de ces adresses est fondamentalement un intervalle des nombres entiers. Toutes les machines non specialisees actuelles appartiennent a cette classe. Nous qualifierons ces machines de "sequentielles".

De telles machines suggerent fortement l'emploi de tableaux pour structurer les donnees. Dans le domaine "non numerique" (ou "ensembliste") on se heurte rapidement a des difficultes bien connues schematisees par la figure ci-dessous:

Représentation d'ensemble (d'entiers) par un tableau ordonné.



Aussi bien pour l'ajout que pour le retrait, il est necessaire de reorganiser.

Aucun probleme concret ne peut se contenter de representations aussi simples. Des qu'un meme ensemble est structure d'au moins deux manieres un mecanisme supplementaire est indispensable. On peut tenter de conserver l'emploi de tableaux, avec des copies du meme ensemble et une representation de la bijection canonique entre les copies. Ajoutons que la simple notion de "liste binaire" introduit deja la necessite de rupture de sequence explicite, puisque la definition recursive necessite une representation concrete.

AINSI NOUS NOUS TROUVONS FACE A LA NECESSITE DE ROMPRE LOGIQUEMENT LA SEQUENTIALITE PHYSIQUE DES DONNEES DANS LA MACHINE. NOUS APPELLERONS "POINTEUR" TOUT MECANISME ABOUTISSANT A CE RESULTAT.

Cette definition du pointeur comme rupture de sequence dans les donnees ne suppose rien ni quant a la nature reelie ou formelle du mecanisme ni quant aux moyens mis en oeuvre pour sa realisation.

La rupture de sequence des donnees, au meme titre que la rupture de sequence du programme, est une notion assez delicate a manier. Cette difficulte a conduit a la notion de structure de controle pour le programme et parallelement a celle de type abstrait pour les donnees. Ces abstractions utiles n'ont pourtant pas donne le coup de grace aux anciennes methodes pour des raisons qui nous semblent fondamentales. Ce mouvement vers l'abstraction s'accompagne en effet d'une tendance contradictoire a la rigidite et a la lourdeur. On peut se contenter de quelques structures simples et tout construire sur elles. Toute situation ne rentrant pas facilement dans le cadre rigide ainsi fixe evolue fatalement vers des solutions lourdes, peu aisees a mettre en

oeuvre, a maintenir, a comprendre. La tentative de definir des structures suffisamment souples et generales aboutit egalement a la lourdeur, cette fois-ci des structures elles-memes.

Des les problemes les plus simples on se heurte a d'importantes difficultes. Nous aborderons dans la section 1.4 le probleme elementaire d'enumerer les arborescences binaires. Voila deja un algorithme avec des structures non classiques, avec des sous-types dynamiquement definis. Une difficulte supplementaire apparait lorsque des objets appartiennent a plusieurs structures. La notion de type abstrait permet de definir efficacement chacune de ces sous-structures, mais leur association sans hierarchie entre elles est un sujet d'etude en soi-meme, ou l'on retrouve en particulier la necessite de traiter des representations.

Il apparait donc que l'accès aux representations doit demeurer possible, sous peine de sterilisation du programmeur. Le seul moyen de liberer le programmeur des contraintes de rigidite et de lourdeur consiste a lui fournir des moyens pour definir lui-meme la representation de ses types abstraits ou pour acceder aux representations fixees par les compilateurs. En particulier la promotion de langages avec types abstraits ne pourra eviter que certains problemes necessitent toujours l'emploi de pointeurs, et nous pouvons conclure que la formalisation de la notion de pointeur n'etait en aucune maniere un combat d'arriere-garde mais qu'il etait au contraire utile de lui definir sa place et son statut dans la programmation, et d'elaborer l'outillage de conception et de mise en oeuvre.

1.3.2. La place des pointeurs dans la programmation.

Les travaux consacres jusqu'ici aux pointeurs [HOA69, HOA73, DEM77] se sont essentiellement interesses a la definition des types associes a la notion de pointeur dans les langages de programmation. Les resultats sont lourds, difficiles a utiliser concretement, ils aident peu le programmeur.

Nous avons prefere renverser le probleme et l'aborder sous l'angle de la programmation structuree. La question a poser est alors: QU'EST-CE QUE LES POINTEURS VONT PERMETTRE DE RAFFINER ET COMMENT ALLONS-NOUS EUX-MEMES LES RAFFINER ? La reponse que nous avons donnee se resume en quelques mots: il est possible et necessaire de definir comme type abstrait de nature particuliere une notion formelle de pointeur, capable en meme temps de représenter les types abstraits les plus divers et d'etre representable dans les langages les plus varies.

Il n'est donc pas question au depart de remplacer les types abstraits par une nouvelle structure, ni a l'inverse de definir une nouvelle semantique de langage. Formaliser utilement les pointeurs ne pouvait se concevoir que comme moyen d'elargir les limites d'emploi des types abstraits et comme definition des outils minima a introduire dans les langages.

1.3.3. Comment programme-t-on avec des pointeurs ?

Programmer avec des pointeurs exige d'abord de programmer de maniere structuree. c'est un point fondamental: les defaults traditionnellement associes a l'emploi des pointeurs sont souvent dans leur realite profonde des defaults de structuration, un manque de discipline dans la mise en oeuvre.

Qu'il y ait ou non specification formelle, le premier temps de programmation doit necessairement consister en un algorithme ecrit a partir des seuls operateurs de la theorie mathematique qui formalise l'objet traite. En particulier aucun choix de representation ne doit etre fait a ce niveau, pour permettre ulterieurement tous les choix sans restriction. Cela impose naturellement la meme contrainte pour les specifications qu'elles soient formelles ou non. En ce qui concerne les problemes justifiant l'emploi des pointeurs, cela signifie concretement que l'algorithme initial est un algorithme de theorie des ensembles finie (c'est-a-dire ou l'on nie l'axiome de l'infini). Ce premier temps de programmation peut parfois necessiter un important travail de remise en forme des algorithmes classiques (voir 7.1.3) ou de remise en cause de certains concepts. Que representent en particulier piles ou files au sens de l'objet reel traite? L'important est ici ce qui est represente, et non la methode classique pour le realiser.

Le deuxieme stade de programmation commence avec la definition de representations pour les ensembles utilises dans l'algorithme de premier niveau. Les deux structures les plus classiques de liste lineaire et de liste binaire ("arbre binaire") correspondent a deux necessites: celle de l'enumeration (balayage) sequentielle et celle de l'amelioration des acces dans une table. Elles ne sauraient pour autant etre les seules, et la perspective d'utilisation de ce que nous appellerons "structures chainees" laisse ouvertes un maximum de possibilites. Notre exemple de la section 1.4 (enumeration des arborescences binaires) illustre parfaitement l'utilisation de cette liberte de choix dans le sens a la fois de la commodite et de l'efficacite. Ce deuxieme stade, qui comprend eventuellement plusieurs raffinements, aboutit a la redaction d'un texte de programme exprime a l'aide des operateurs primitifs sur les types abstraits utilises.

Ce deuxieme programme ainsi acheve peut parfois etre raffine assez rapidement en un programme correct, demontre et comprehensible. C'est en particulier le cas lorsqu'on envisage l'emploi des pointeurs avec un langage disposant lui-meme d'un mecanisme de pointeurs. Dans d'autres cas, un troisieme stade explicite pourra etre indispensable. Ce sera le cas general lorsqu'il faudra utiliser des tableaux dans le langage. En effet la troisieme forme de l'algorithme est trop differente du programme definitif pour se permettre l'economie de son ecriture explicite. Naturellement on ne peut jamais eviter la fourniture des conditions du raffinement soit comme commentaire du programme, soit dans le rapport d'accompagnement.

Le passage d'une forme exprimee en types abstraits a une forme en structure chainee n'est pas un simple changement de formalisme. Comme tout raffinement il exprime des choix et les met en oeuvre. Ces choix peuvent se resumer schematiquement comme des allocations de "place" pour les diverses informations necessaires aux types abstraits. C'est a ce niveau qu'on determine quelles informations doivent etre conservees et ou, et quelles autres seront recalculées et comment. C'est aussi a ce niveau qu'un ensemble dynamique de listes doit etre etudie plus completement pour aboutir a une definition statique de l'emplacement des informations.

Comme nous l'avons dit plus haut, il est parfois possible d'eviter l'ecriture d'un programme explicite en structure chainee. A l'inverse l'ecriture precise de la definition de structure est un document de haute importance pour la relecture ou la modification du programme obtenu. Cette definition ne peut se reduire aux declarations de types correspondantes. Elle doit comprendre tout ce qui definit le "squelette" (intuitivement la partie statique - voir details chapitre 3), avec un maximum de renseignements sur les moules (intuitivement ce qui supporte les informations elementaires) et l'axiomatique avec sa signification en termes de types abstraits. On n'oubliera pas l'indication pour les elements de chaque type des "moules" qui les referencent, en quelque sorte la reciproque des moules. L'experience montre que ce compte de references theorique, tel qu'il resulte des axiomes, est un outil utile.

La demonstration d'un programme ecrit en structures chainees n'est souvent que la demonstration d'adequation d'un raffinement, ce qui est en general simple. Dans certains cas (voir l'exemple de 1.4) il s'agit d'une structure originale qui demande sa propre demonstration. Dans ces preuves on devra distinguer la structure donnee non modifiee, ou toutes les methodes classiques de construction, transformation et preuve sont applicables, de la construction de la structure nouvelle, qui exige l'application stricte de la formalisation.

Si on accepte la discipline d'application de ces divers elements de methode, l'utilisation des pointeurs dans la programmation n'est pas fondamentalement plus dangereuse ni plus inefficace que d'autres methodes de representation, a complexite de sujet comparable. Les difficultes et les erreurs proviennent plus de l'inadaptation des outils et du refus de discipline de structuration que de tout autre facteur.

1.4. PRESENTATION D'UN EXEMPLE

Arrives a ce point il nous semble souhaitable d'introduire un exemple pouvant servir a materialiser nos developpements ulterieurs. Il est ici inutile de reprendre la construction traditionnelle ou non de l'un quelconque des types abstraits de base. Nous ne prenons a ce point de l'expose qu'un exemple d'ecole, ce qui importe etant la methode et ses difficultes d'application. Nous exposerons ulterieurement (chapitre 7) une experimentation en vraie grandeur.

1.4.1. Enonce et formalisation.

Nous nous fixons pour but de construire un programme d'enumeration des arborescences binaires.

Cet enonce informel est d'une grande ambiguite. Precisons-le tout de suite en disant qu'il faut comprendre "arborescence" et "binaire" dans leur sens le plus strict de la theorie des graphes. Il est donc question de graphes orientes sans cycle ayant une racine, laquelle est alors unique. Le qualificatif "binaire" signifie naturellement que chaque sommet a au plus deux successeurs, sans consideration d'ordre entre ceux-ci. Une telle structure differe notablement de ce qu'on entend habituellement par abus de langage sous le meme nom.

Terminons de preciser l'enonce en indiquant que l'ordre d'enumeration est indifferent. Il suffit de garantir l'exhaustivite et l'absence de repetitions.

1.4.1.1. Analyse du probleme.

Le graphe vide n'est pas une arborescence binaire. Le graphe a un seul sommet et sans arc est l'unique arborescence binaire a un sommet.

De plus toute arborescence binaire a au moins deux sommets peut se decomposer en sa racine et une ou deux arborescences "filles". Si l'arborescence "mere" est binaire, il en sera de meme pour sa ou ses filles. Dans la mesure ou il n'y a pas d'ordre entre les successeurs, cette decomposition est unique.

1.4.1.2. Synthèse.

On peut donc enumerer les arborescences binaires de n sommets ($n \geq 1$) des que l'on sait enumerer celles a moins de n sommets. puisqu'on connait toutes les arborescences binaires a un sommet, la question de l'exhaustivite est theoriquement reglee.

Il reste a eliminer d'eventuelles repetitions. Si l'enumeration initiale est sans repetition, on n'en introduira pas dans le cas 1, mais dans le cas 2 les situations ci-dessous sont indiscernables.



La solution la plus apparente consiste a introduire un ordre total sur les arborescences deja traitees. N'interviennent alors dans l'enumeration que les couples (A,B) pour lesquels $A \neq B$. Dans notre cas un ordre total peut etre obtenu aisement en considerant $A < B$ si et seulement si on est dans l'un des cas exclusifs suivants

- A a strictement moins de sommets que B
- A et B ont le meme nombre de sommets et A est creee avant B dans l'enumeration.

1.4.1.3. Premiere construction de programme.

Nous pouvons a ce point ecrire une premiere version de l'algorithme de calcul des arborescences binaires a $N+1$ sommets a partir de celles qui ont au plus N sommets.

Ce programme comme les suivants est ecrit en un pseudo-langage ou seront utilisees les structures de controle classiques ou non qui apparaitront necessaires, et les instructions de base correspondant aux fonctions elementaires sur les objets manipules.

```

:  algorithme UNSOMMETDEPLUS ( N : entier positif )
:  balayer par A les arborescences a N sommets faire
:      UTILISER ( ARB01 ( A ) )
:  pour M entre 1 compris et N/2 exclu faire
:      balayer par A les arborescences a N-M sommets faire
:          balayer par B les arborescences a M sommets faire
:              UTILISER ( ARB02 ( A , B ) )
:  si N pair alors
:      balayer par A les arborescences a N/2 sommets faire
:          balayer par B les arborescences a N/2 sommets
:              a partir de A compris faire
:                  UTILISER ( ARB02 ( A , B ) )
:  fin.

```

Dans ce programme ARB01 et ARB02 sont les fonctions "mathematiques" qui construisent une arborescence dans les deux cas possibles (un seul ou deux successeurs pour la racine). Bien entendu UTILISER a exactement le role d'une instruction commandee dans l'enumeration ainsi construite.

1.4.2. Resolution "informatique" du probleme.

Le "programme" ci-dessus est encore fondamentalement mathematique. Il ne contient aucune notion de representation ni de partage de "place" dans le temps. Il se contente d'integrer sous la forme d'un "balayage" l'existence d'ordres totaux.

Nous raffinerons ce programme en introduisant deux choix qui semblent tres naturels dans ce contexte. Une arborescence sera representee par un couple dont les elements sont soit une arborescence soit un objet distingue appele "vide". Nous avons ainsi avec (vide,vide) l'arborescence a un seul sommet, ARB01(A) est (A,vide) et ARB02(A,B) est (A,B). Nous garantirons l'absence de circularites dans cette structure par l'existence de la fonction "nombre de sommets" compatible (au sens des graphes sans circuit).

Le second choix, suggere par l'existence d'ordres totaux, est la representation par une liste lineaire de chacun des ensembles d'arborescences binaires du meme nombre de sommets.

Voici le programme ainsi raffine tel qu'il semble souhaitable de l'obtenir. Nous discuterons ensuite les problemes de definition des types des variables utilisees.

```

:  algorithme UNSOMMETDEPLUS ( N : entier positif )
:  creer liste vide L(N+1)
:  A := premier de L(N)
:  tant que A defini faire
:      ajouter ARB(A,vide) dans L(N+1) en queue
:      A := suivant de A dans L(N)
:  fin tant que
:  pour M entre 1 compris et N/2 exclu faire
:      A := premier de L(N-M)
:      tant que A defini faire
:          B := premier de L(M)
:          tant que B defini faire
:              ajouter ARB(A,B) dans L(N+1) en queue
:              B := suivant de B dans L(M)
:          fin tant que
:          A := suivant de A dans L(N-M)
:      fin tant que
:  fin pour
:  si N pair alors
:      A := premier de L(N/2)
:      tant que A defini faire
:          B := A
:          tant que B defini faire
:              ajouter ARB(A,B) dans L(N+1) en queue
:              B := suivant de B dans L(N/2)
:          fin tant que
:          A := suivant de A dans L(N/2)
:      fin tant que
:  fin si
:  fin algorithme

```

Dans cette version "UTILISER" a ete raffine explicitement en une addition dans la nouvelle liste, tandis qu'une nouvelle fonction ARB sert a construire le couple forme de ses deux arguments.

1.4.3. Problemes de types et de representation.

Ayant le programme ci-dessus, attachons-nous a definir les types de ses diverses variables. Nous aurons besoin d'un type "arborescence" qui pourrait s'abstraire comme suit:

```

:  = un ensemble E, un objet isole v et  $F = E \cup \{v\}$ 
:  = trois fonctions ar :  $F * F \rightarrow E$    pp, sp :  $E \rightarrow F$ 
:  = les axiomes ar(pp(x),sp(x))=x
:  =      pp(ar(x,y))=x
:  =      sp(ar(x,y))=y

```

Nous sommes maintenant capables de definir une "liste d'arborescences" ce qui fournit un type abstrait utilisable pour A, B et chaque $L(N)$.

Une telle definition de type, pour correcte qu'elle soit, ne saurait reellement nous satisfaire. Elle signifie concretement que A et B peuvent parcourir l'univers entier et que les $L(N)$ sont des objets structures de meme nature. Tout cela ne nous fournit pas l'outillage de verification qui nous serait necessaire.

Si nous reprenons le programme nous constatons qu'une des listes $L(K)$ a un statut particulier puisqu'elle est modifiable mais non les autres. Cette liste distinguee $L(N+1)$ depend de l'argument d'entree N de l'algorithme. Nous constatons que d'une iteration a l'autre les variables de commande A et B sont astreintes a une liste precise mais determinee dynamiquement.

Un controle de type totalement efficace devrait pouvoir prendre en compte tres finement l'emploi des variables. Il serait necessaire d'astreindre A a reperer un element dans la liste $L(N)$, $L(N-M)$, $L(N/2)$ respectivement, tout en etant admis indistinctement dans les trois appels de ARB. De meme il est imperatif que A et B soient de types compatibles pour le cas "N pair" et incompatibles dans le reste du programme.

Il est clair qu'on ne trouvera pas de solution avec les types a ensemble d'operateurs restreint [LIS78] dans la mesure ou les ensembles d'operateurs ainsi utilises seraient dynamiques (premier, suivant, ajouter). De maniere generale toute tentative de definition de type servant a quelque chose aboutit a une solution essentiellement dynamique. Il n'y a pas de controle de type possible sans un demontreur de theoremes non trivial.

Nous nous trouvons ici en tout etat de cause avec un ensemble de listes $L(1)$, $L(2)$, etc... potentiellement infini. Nous savons qu'un tel ensemble n'est pas representable, sauf en definissant une limite de taille ou en utilisant des particularites de la structure. Tout essai de choix automatique de representation devra donc commencer par demontrer un theoreme qui ici pourrait etre: "chaque arborescence appartient a une liste $L(N)$ et une seule, celle pour laquelle N est le nombre de sommets". Ce theoreme est indispensable pour rendre possible une representation, et le choix associe au raffinement suivant portera sur cette necessite de représenter une multiplicité de listes.

Nous envisagerons ultérieurement un tel raffinement, en notant des maintenant qu'il tend à détruire dans le programme la distinction entre les diverses listes, le "chainage" "suivant" étant unique et seule la "tête" caractérisant la liste utilisée. De manière générale une telle situation se rencontrera chaque fois qu'un ensemble devra être partitionné (par exemple les arcs d'un graphe en fonction de leur origine).

1.4.4. Conséquences de l'étude rapide de cet exemple.

Cet exemple, malgré sa simplicité, souligne l'existence de limites intrinsèques à l'emploi des types abstraits, et de problèmes importants liés à leur représentation.

De manière générale nous avons rencontré trois grandes catégories de difficultés:

- la pleine puissance des contrôles de type nécessite l'emploi de définitions dynamiques;
- l'emploi dans une même structure de deux types abstraits introduit des propriétés nouvelles pour les types initiaux;
- l'existence de matérialisations multiples pour un même type crée des contraintes de représentation.

Dans les divers cas on se heurte à une insuffisance de l'outillage fourni par les types abstraits, outillage qu'il ne semble pas possible d'étendre suffisamment en conservant les mêmes principes de base. Il est indispensable d'étudier en tant que telles les représentations (aussi formelles que possible) des types abstraits classiques, et parmi celles-ci les pointeurs ont nécessairement une grande place.

1.5. PLAN COMMENTÉ.

La définition d'un ensemble de types d'un langage de programmation définit en quelque sorte la structure libre que ces types permettent de réaliser. Parmi les objets complexes possibles, seuls quelques uns correspondent aux réalités à représenter. Il est donc indispensable de définir aussi l'axiomatique de la structure utile.

Dans le cas qui nous occupe ici, l'existence de fonctions partielles complique la formalisation de ces axiomes. Nous avons donc été conduit à définir dans le cadre du calcul des prédicats du premier ordre une notion adaptée. Bien entendu on n'utilisera qu'exceptionnellement cette construction, de la même manière qu'un mathématicien n'écrit pas ses démonstrations en calcul des prédicats. Elle sert en premier lieu de fondement, et en second lieu de voie de recours lorsque l'auteur et les lecteurs des démonstrations sont en désaccord sur leur validité.

La definition de ce "calcul des predicats adapte aux structures chainees" fera l'objet du chapitre 2 suivi de son application a l'exemple de 1.4 que nous utiliserons a chaque etape comme application.



Le chapitre 3 definissant la notion de "structure chainee" est le coeur de ce travail. Cette notion est en effet la formalisation, dont nous avons souligne la necessite, qui permet a la fois la synthese des proprietes des types abstraits et de leurs representations.

Cette formalisation comprend cinq elements: la structure libre appelee "squelette", les realisations d'une telle structure appelees "constructions", l'axiomatique, les operateurs sur les constructions et les operateurs sur le squelette. Nous envisageons parmi les applications une notion de pointeur "flottant" pendant les destructions, en la justifiant dans notre cadre formel par une operation de passage au quotient.



Les structures chainees ne constituent pas une formalisation des pointeurs dans les langages, mais un type abstrait particulier ou mieux un constructeur de types abstraits adaptes aux problemes les plus divers.

Il faut donc realiser ces types, dans un maximum de langages de programmation. Concretement on se retrouve avec deux grandes classes de representations, avec les tableaux du langage ou avec ses pointeurs. L'une et l'autre montrent les inadaptations des langages a la programmation avec pointeurs.

Nous aborderons la question des representations permanentes hors memoire centrale, qui necessitent a elles seules une etude specifique.

Ces questions feront l'objet de deux chapitres, le quatrieme consacre aux representations en general, et le sixieme ou nous ferons diverses propositions pour adapter les langages aux besoins exprimes precedemment.



Deux structures de base constituent la quasi-totalite des applications: la liste lineaire et la liste binaire (ou arbre binaire). Nous etudierons ces deux notions, d'abord la liste lineaire comme systeme formel, ensuite l'une et l'autre comme structure chainee.

Dans ce processus, si l'utilisation de la definition formelle peut etre utile, elle ne permet pas de definir les notions de sous-liste ou d'insertion en un endroit determine. Pour toute utilisation informatique, il faut y adjoindre une notion de place qui devient en structure chainee un "pere". Nous ne faisons avec cette adjonction qu'introduire la distinction entre l'objet et l'information qu'il porte, ce qui semble assez legitime.

Nous n'avons pas defini de notion formelle de liste binaire en raison d'une impossibilite qui nous semble essentielle: la non-existence de cycles dans la structure n'est definissable que soit au deuxieme ordre, soit en introduisant une fonction reciproque des chainages, laquelle n'est applicable qu'a une sous-structure donc au travers ici encore d'une notion de place.



Nous nous sommes efforce au long de ce travail d'en exposer une application concrete. L'application elementaire de la section 1.4 se poursuit au travers des chapitres fondamentaux pour les illustrer.

Mais nous ne pouvions pas nous contenter d'un exemple scolaire. Nous avons exprime plus haut cette idee que l'association non hierarchique de structures entraînait par elle-meme des difficultes nouvelles. Nous avons donc traite completement un probleme concret de taille significative. Le chapitre 7 decrira l'essentiel du processus et une annexe fournira quelques extraits du programme et des resultats, un rapport technique complet devant etre edite ulterieurement.



En guise de conclusion nous pouvons dire que les travaux exposes dans les chapitres 2 a 7 appellent des extensions dans trois directions principales: la realisation d'outils appropries, l'etude des particularites des structures permanentes, les types abstraits.

Nous ne concevons pas comme un but acceptable la conception et la mise en oeuvre d'un nouveau langage adapte specifiquement a la realisation des structures chainees. Ayant toujours en vue la programmation concrete, nous estimons preferable d'oeuvrer a la mise en oeuvre dans les langages les plus courants d'un minimum d'outillage, tel qu'il a ete evoque au chapitre 6. Compte tenu du role de Pascal comme base des langages nouveaux et de son extension dans le domaine industriel, la realisation d'une extension "chainee" de Pascal semble envisageable.

Ces travaux definissent une formalisation fortement independante des representations. Parmi celles-ci nous avons laisse de cote les representations permanentes, disons pour fixer les idees sur disque en acces direct. Or celles-ci posent des problemes assez complexes d'accès et de fiabilite, a l'interieur du meme cadre formel. Entre la representation simple et inefficace qui lit et eventuellement reecrit

pour chaque reference a un pointeur, et la representation la plus generale qui lit et reecrit en fonction de l'evolution des variables, il y a un moyen terme consistant a deduire de l'algorithme un optimum d'accès. C'est l'etude de ce moyen terme que nous proposons comme deuxieme extension de nos travaux.

Enfin les structures chainees ont ete construites sur la base d'une analyse critique des types abstraits. Il est donc necessaire d'assurer le mouvement de retour pour apporter aux types abstraits et aux langages les utilisant des solutions acceptables aux difficultes qui sont apparues, et ceci constitue une troisieme voie d'extension.

ADAPTATION DU CALCUL DES PREDICATS A LA FORMALISATION DES POINTEURS

2.1. IDEE DIRECTRICE

Dans le chapitre d'introduction nous avons souligne comme elements essentiels definissant les pointeurs l'existence d'objets de types differents lies entre eux par des fonctions partielles unaires, lesquelles doivent respecter certaines conditions.

Pour definir ces conditions d'une maniere suffisamment formelle nous allons definir un calcul des predicats adapte a notre probleme, dont nous montrerons qu'il n'est qu'un cas particulier du calcul des predicats du premier ordre classique.

2.2. CALCUL DES PREDICATS AVEC TYPES D'OBJETS

Nous supposerons connue la notion de calcul des predicats du premier ordre ainsi que les notions de base de sa construction: variables, symboles fonctionnels ou relationnels, termes, formules atomiques, formules, formules closes, quantificateurs, realisations, valeur d'une formule, axiomes,... [KRE67].

Le calcul des predicats avec types d'objets est peut-etre moins bien connu et une introduction nous semble utile, ne serait-ce que pour souligner les aspects a modifier pour atteindre notre but.

2.2.1. Definitions generales.

Un calcul des predicats du premier ordre avec types d'objets (que nous abregerons pendant ce chapitre en OPTO) se construit a partir de:

- un ensemble informel fini T de types
- pour chaque type t , un ensemble V_t de variables et un ensemble C_t de symboles de constantes.
- pour chaque entier non nul n , un ensemble $R^{(n)}$ de symboles relationnels n -aires non types.

- un ensemble S de symboles relationnels types, pour chacun desquels est definie une suite finie de types appelee profil du symbole.

Tous ces ensembles sont deux a deux disjoints.

Il n'y a pas de symboles fonctionnels autres que les constantes. Ainsi il n'y a comme termes que les variables et les symboles de constantes, chacun canoniquement muni d'un type.

Les formules atomiques sont de l'une des formes suivantes:

- $r a_1 \dots a_n$ ou r est dans $R^{(n)}$ et $a_1, \dots, a_n$ sont des variables ou des symboles de constantes
- $s a_1 \dots a_n$ ou s est dans S , de profil $(t_1, \dots, t_n)$ et ou chaque a_i est un terme de type t_i .

Les formules se construisent comme dans le calcul non type avec les connecteurs $\neg$, $\vee$ et $\forall x$ ou x parcourt la reunion des V_t . Les abreviations $\wedge$, $\rightarrow$, $\leftrightarrow$, $\wedge x$ ainsi que les notions de variable libre et de formule close se definissent aussi de la meme facon.

Tout ceci nous definit le langage des formules du CPTO considere.

Une realisation d'un tel langage se compose:

- pour chaque type t d'un ensemble E_t appele ensemble de base du type t pour cette realisation
- pour chaque symbole relationnel non type n -aire r d'une partie $\tilde{r}$ de $(\bigcup_{t \in T} E_t)^n$
- pour chaque symbole s de S , de profil $(t_1, \dots, t_n)$, d'une partie $\tilde{s}$ de $\prod_{i=1}^n E_{t_i}$
- pour chaque c de C_t , d'un element $\tilde{c}$ de E_t

Avant de continuer, il est utile de definir des notations permettant de simplifier ulterieurement certaines ecritures.

Nous noterons E_* la reunion des E_t , $VL(F)$ l'ensemble des variables libres de la formule F . L'ensemble $VL(F)$ est ainsi une partie de V_* reunion des V_t . A tout ensemble Z de variables nous associerons l'ensemble $W(Z)$ suivant:

$$W(Z) = \prod_{t \in T} E_t^{(Z \cap V_t)}$$

En d'autres termes $W(Z)$ est l'ensemble des fonctions f de Z dans E_* qui satisfont la contrainte de type "si la variable z est de type t , alors la valeur $f(z)$ est dans E_t ". Cette definition apparemment lourde permet de definir la valeur d'une formule F comme une partie de $W(VL(F))$.

La notation W etant donnee, la valeur d'une formule dans la realisation se definit classiquement par induction structurelle.

- $Val(a_1...a_n)$ se compose des elements f de $W(VL)$ tels que $\langle x_1,...,x_n \rangle \in \bar{r}$ ou x_i est $\bar{c}$ si a_i est la constante c et $f(a_i)$ si c est une variable

- $Val(\neg F) = W(VL(F)) - Val(F)$

- $Val(F \vee G) = (Val(F) * W(X)) \cup (Val(G) * W(Y))$ ou nous avons note $X = VL(G) - VL(F)$ et $Y = VL(F) - VL(G)$

- $Val(\forall x F) \cup \{x\} = \{ \delta \mid (\exists \lambda \in W(VL(F))) \lambda \upharpoonright W(VL(F)) = \delta \text{ et } \lambda \in Val(F) \}$ ou $a \upharpoonright b$ designe la restriction a b de la fonction a .

La complication n'est qu'apparente. Intuitivement la signification est analogue au cas non type:

- $Val(\neg F)$ est le complementaire de $Val(F)$

- $Val(F \vee G)$ est la reunion de $Val(F)$ et $Val(G)$

- $Val(\forall x F)$ est la projection selon x de $Val(F)$

Les notions de realisation satisfaisant une formule ainsi que d'axiome et de realisation d'un systeme d'axiomes sont analogues au calcul des predicats sans type.

2.2.2. Definition d'un calcul type dans un calcul non type.

Le CPTO differe des calculs sans type essentiellement par l'absence de symboles fonctionnels et l'introduction de types d'objets. Il est important de montrer qu'on peut introduire des types dans le formalisme non type, et inversement des fonctions dans le calcul type. Ainsi les deux calculs sont equivalents, et on les utilisera en fonction de leur commodite par rapport au probleme pose. Cette equivalence nous permettra ulterieurement de definir notre calcul adapte aux pointeurs sur le formalisme type en affirmant immediatement la possibilite de le definir en calcul non type, avec cependant des expressions tres alourdies.

Considerons donc un CPTO avec les notations de la section precedente. Nous definissons un calcul des predicats non type (que nous abregerons par commodite en CPNT) comme suit:

- les variables du CPNT sont celles du CPTO

- les symboles fonctionnels du CPNT sont les symboles de constantes du CPTO avec la meme arite zero

- les symboles relationnels du CPNT sont les symboles relationnels, tant types que non types, du CPTO, chacun conservant son arite

- on ajoute des symboles relationnels unaires Q , un pour chaque type.

Avec cette definition le langage des formules du CPTO est contenu dans celui du CPNT.

Considerons maintenant un ensemble E de formules du CPTO de depart et un modele L de E. Ce modele est une realisation du CNPT associe pour toute interpretation des symboles Q_t . Dans la suite nous

les interpreterons par $\bar{Q}_t = E_t$, c'est-a-dire informellement " $Q_t x$ " exprime " x est de type t". Appelons M la realisation ainsi construite. Nous avons alors:

- $Val_L(rx_1...x_n) = Val_M(rx_1...x_n) \subseteq E^n$
- $Val_L(sx_1...x_n) = Val_M(sx_1...x_n) \subseteq W(x_1,...,x_n) \subseteq E^n$

Transformons maintenant toute formule f typee en une formule $f^{\sim}$ non typee en remplaçant par induction structurelle toute construction " $\forall x A$ " par " $\forall x_t Q x A$ " ou t est le type du symbole x. Pour un ensemble A de formules, nous noterons tout naturellement $A^{\sim}$ l'ensemble des $f^{\sim}$ lorsque f parcourt A.

La propriete fondamentale est que tout modele dans L d'un axiome f se transforme en un modele M de la formule

$$f^{\sim} \wedge \bigwedge_{x \in V(L)} Q_{type(x)} x$$

Cela se demontre classiquement par induction. C'est un resultat elementaire de theorie des ensembles pour la negation et la disjonction. On le verifie aisement pour les formules atomiques:

$$Val_M(\bigwedge_{x \in V} Q_{type(x)} x) = W(V)$$

$$Val_L(sx_1...x_n) = Val_M(sx_1...x_n) \subseteq W(x_1,...,x_n)$$

$$Val_M(rx_1...x_n) = \{ \delta \in E^n \mid \langle \delta x_1, \dots, \delta x_n \rangle \in \bar{r} \}$$

$$Val_L(rx_1...x_n) = \{ \delta \in W(x_1,...,x_n) \mid \langle \delta x_1, \dots, \delta x_n \rangle \in \bar{r} \}$$

La demonstration est plus complexe pour les quantifications. Son ecriture se simplifie notablement en introduisant des notations adaptees. Pour la formule $A(x_1,...,x_n)$ et la variable x nous noterons:

- $V^+ = VL(A)$ $W^+ = W(V^+)$ $D = E^{V^+}$
- $V^- = VL(A) - \{x\}$ $W^- = W(V^-)$ $D = E^{V^-}$
- $K = D^- * W(x)$ partie de D^+ contenant W^+

$$- G = \{ \delta \in D \mid (\exists \lambda \in \text{Val}_M(A^{\sim}) \cap K) \quad \lambda \vdash D = \delta \}$$

$$- H = \{ \delta \in W \mid (\exists \lambda \in \text{Val}_L(A)) \quad \lambda \vdash W = \delta \}$$

Avec ces notations il nous faut demontrer $G \cap W = H$ a partir de l'hypothese d'induction $\text{Val}_L(A) = \text{Val}_M(A^{\sim}) \cap W$. En l'introduisant dans H on obtient

$$- H = \{ \delta \in W \mid (\exists \lambda \in \text{Val}_M(A^{\sim}) \cap W^+) \quad \lambda \vdash W = \delta \}$$

Calculons $G \cap W$.

$$- G \cap W = \{ \delta \in W \mid (\exists \lambda \in \text{Val}_M(A^{\sim}) \cap K) \quad \lambda \vdash D = \delta \}$$

Mais un tel λ appartient a K donc $\lambda(x) \in E$. Puisque $\lambda \vdash D = \delta$ element de W, on a en plus $\lambda(x_1) \in E_{\text{type}(x_1)}$. Donc tout λ convenable appartient a W. En

consquence $\lambda \vdash D = \lambda \vdash W$. Il resulte de ces deux remarques qu'on a

$$- G \cap W = \{ \delta \in W \mid (\exists \lambda \in \text{Val}_M(A^{\sim}) \cap W^+) \quad \lambda \vdash W = \delta \}$$

qui n'est autre que H, ce que nous cherchions.

Nous avons donc demontre:

$$\text{Val}_M(A^{\sim}) \cap W(VL(f)) = \text{Val}_L(f)$$

Il en resulte que tout resultat en CPTD peut aussi s'obtenir par un CPNT convenable, c'est-a-dire que l'emploi d'un calcul des predicats avec types d'objets se justifie chaque fois que cela apporte une simplification d'usage.

2.2.3. Introduction de fonctions dans le calcul type.

Les calculs des predicats types ont le defect important de ne pas comporter de symboles fonctionnels. Leur introduction complique en effet notablement la definition du langage des formules. Definir un terme type n'est guere simple et sa realisation encore moins. Il est preferable d'operer de maniere indirecte en definissant les symboles fonctionnels comme abreviations associees a certains symboles relationnels particuliers. Dans notre cas nous beneficierons du fait que les symboles fonctionnels fondamentaux sont unaires.

Etant donne un symbole relationnel s, n-aire, de profil $\langle t_1, \dots, t_n, u \rangle$, nous introduirons les formules:

$$- \text{RSD}_n(s) : \lambda x_1 \dots \lambda x_n \quad \forall y \quad s x_1 \dots x_n y$$

$$- \text{RID}_n(s) : \lambda x_1 \dots \lambda x_n \quad \lambda y \quad \lambda z \quad (s x_1 \dots x_n y \wedge s x_1 \dots x_n z) \rightarrow y = z$$

$$- \text{RFD}_n(s) : \text{RSD}_n(s) \wedge \text{RID}_n(s)$$

Dans ces trois schemas les variables $x_1, \dots, x_n, y, z$ ont les types respectifs $t_1, \dots, t_n, u, u$ qui rendent les ecritures coherentes.

Etant donne un symbole relationnel s comme precedemment, on lui associe un symbole fonctionnel g tel que pour toute formule $A(x)$ l'expression $A(gx_1 \dots x_n)$ soit consideree comme abreviation pour " $\forall y (A(y) \wedge s x_1 \dots x_n y)$ ". Le profil de s sera appele profil de g .

Avec ces definitions, dans toute realisation satisfaisant RFD (s) (respectivement RID (s)), la realisation $\bar{s}$ de s est une fonction (respectivement fonction partielle) de $E_{t_1} * \dots * E_{t_n}$ dans E_u . En tant qu'axiome, $RID_n(s)$ exprime notamment l'unicite de y dans l'expression $A(gx_1 \dots x_n)$.

Nous utiliserons abondamment cette construction dans la section suivante. Elle permet aussi de realiser la conversion inverse de celle du paragraphe precedent, c'est-a-dire definir un calcul des predicats sans type dans le formalisme des calculs types.

2.3. CALCUL DES PREDICATS POUR LES STRUCTURES CHAINEES

Nous venons d'evoquer la possibilite d'introduire des symboles fonctionnels dans un calcul des predicats avec types d'objets (CPTO). Dans une situation ou les fonctions ont une importance fondamentale, ce qui est le cas pour les structures que nous envisageons d'introduire, un tel formalisme est d'une lourdeur excessive. Il est donc indispensable de definir un formalisme nouveau, definissable en termes de CPTO (donc, au besoin, de calcul non type) mais surtout d'utilisation commode.

2.3.1. Elements de base

Nous allons definir parallelement deux formalismes dont l'un contient l'autre, que nous appellerons en abrege CPPS et CPPE (intuitivement calcul des predicats adapte aux pointeurs, le premier "strict", le secons "etendu").

Dans un CPPS on distingue parmi les types trois categories:

- les R-types en nombre fini quelconque (non nul)
- les I-types en bijection avec les precedents. Si t est un R-type, on notera en general $t^\sim$ le I-type associe.
- un unique N-type note " th ".

Le CPPE aura en plus une quatrieme categorie:

- des S-types en nombre fini quelconque.

Pour s'eloigner le moins possible des notations des CPTD, nous noterons T_R, T_I, T_N, T_S les ensembles de types et T leur reunion.

Il n'y a pas de constantes.

De meme qu'il existe des contraintes sur les types, il existe en CPPS et CPPE des contraintes sur les profils. Seuls seront autorises les profils suivants:

- $\langle tr \rangle$	note	$T(tr)$
- $\langle tr', tr'' \rangle$	note	$C(tr', tr'')$
- $\langle tr, tr^\sim \rangle$	note	$I(tr)$
- $\langle tr, th, tr \rangle$	note	$M(tr)$
- $\langle ti, \dots, ti \rangle$ "k fois"	note	$V_k(ti)$
- $\langle ts, \dots, ts \rangle$ "k fois"	note	$V_k(ts)$
- $\langle th, \dots, th \rangle$ "k fois"	note	$V_k(th)$
- $\langle ts_1, \dots, ts_n, tr \rangle$	note	$S(ts_1, \dots, ts_n, tr)$

Dans ces ecritures, les divers tr , ti et ts designent respectivement des R-types, I-types et S-types.

Il existe des symboles relationnels particuliers, un symbole d'egalite de profil $\langle t, t \rangle$ pour chaque type t , et deux symboles speciaux zer de profil $V_1(th)$ et suc de profil $V_2(th)$.

En plus des constructeurs de formules habituels $\neg$, $\vee$ et $\forall x$, il existe deux constructeurs particuliers def et mul.

Etant donne une suite $\langle \langle t_i, u_i \rangle, \dots \rangle$ de profils et la suite correspondante $\langle f_1, \dots, f_k \rangle$ de symboles, nous dirons que $\langle f_k, \dots, f_1 \rangle$ est une composition coherente lorsque:

- tous les types t_i et u_i sont des R-types
- $u_j = t_{j+1}$ pour $j=1, \dots, k-1$

Une composition coherente sera dite fermee lorsque en plus $u_k = t_1$. Dans ce cas tous les types concernes sont des R-types des que les profils sont parmi les profils autorises.

Il n'y a pas de symboles relationnels non types.

Les symboles de profil $I(tr)$ verifient l'axiome RFD qui leur est associe. Les symboles de l'un des profils $T(tr)$, $C(tr', tr'')$ et $S(\dots, tr)$ verifient le RID associe. Les symboles de profil $M(tr)$ verifient un axiome particulier AFM defini plus loin.

2.3.2. Langage de formules

Un CPFS (respectivement CPPE) est un CPTO qui respecte les contraintes exposees au paragraphe precedent avec une definition convenable de *def* et *mul* comme abreviations.

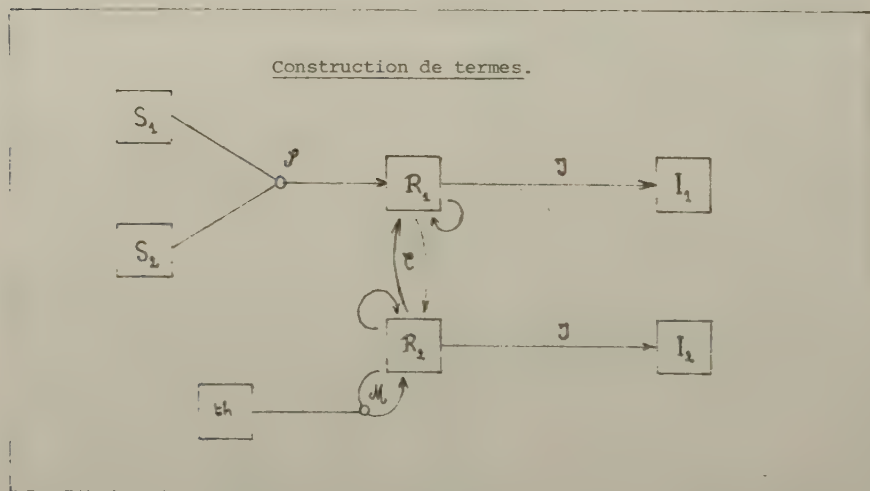
Il est plus commode de definir directement un langage adapte. La notion de variable est inchangee. Les symboles des profils **T**, **C**, **I** et **S** sont traites comme fonctionnels, ceux des profils **V** comme relationnels et ceux des profils **M** ont un traitement special.

On appellera termes de type *t*:

- les variables de type *t*
- les expressions $fx_1...x_n$ ou *f* est de profil $(t_1,...,t_n,t)$ parmi les groupes **T**, **C**, **I**, **S** et ou $x_1,...,x_n$ sont des termes de type $t_1,...,t_n$
- les expressions $mul\ g\ n\ x$ ou *g* est une composition coherente fermee de type *t*, *n* une variable de type *th*, et *x* un terme de type *t*.

En particulier un terme non reduit a une variable est impossible parmi les S-types et le N-type. Un terme d'un I-type *t* autre qu'une variable est un symbole de profil **I**(*t*) applique a un terme de type *t* (R-type).

Un schema illustre intuitivement les diverses possibilites de construction de termes.



On appellera formules atomiques:

- les expressions $rx_1 \dots x_k$ ou r est un symbole relationnel de profil $\nabla_{k,1}(t)$ et ou tous les x_i sont de type t_1 (ceci comprend les egalites notees plus frequemment $x_1=x_2$)
- les expressions $\text{def } x$ ou x est d'un R-type, avec $\text{Val}(\text{def } x) = E_{\text{type}(x)}^{(x)}$ pour toute variable x .

En particulier il n'existe pas de relations entre des objets de types differents. Le seul symbole relationnel sur un R-type est l'egalite.

La construction des formules utilise comme pour les calculs precedents les connecteurs $\neg, \vee$ et $\forall x$. On emploiera aussi avec la meme definition les abreviations $\wedge, \rightarrow, \leftrightarrow, \wedge x$ ainsi que la notation infixee parenthesee.

2.3.3. Axiomatique minimale

Une realisation d'un CPPE ou CPPS ne sera pas n'importe quelle realisation du CPTO qu'on peut canoniquement lui associer. Elle devra satisfaire une axiomatique minimale composee:

- des axiomes de l'egalite
- des axiomes RID(f) pour tous les symboles f d'un des profils C, T, S, M .
- des axiomes RFD(f) pour tous les symboles f de profil I .
- des axiomes de l'arithmetique du premier ordre pour le type th
- des axiomes AFM definis a l'alinéa suivant.

Nous avons introduit un profil $M(tr) = (tr, th, tr)$ assez special dans son utilisation, puisqu'il intervient uniquement en liaison avec le symbole additionnel mul . Pour chaque symbole c de profil $M(tr)$ nous noterons c^* une certaine composition coherente fermee de type tr univoquement associee a c . Pour toute composition coherente fermee $g = (h_1, \dots, h_k)$ de type tr et tout symbole f de profil $M(tr)$ nous noterons AFM(f, g) la formule suivante, exprimee par rapport aux symboles consideres comme relationnels:

- AFM(f, g) : $\forall \delta \text{ si } \text{zer}(\delta) \text{ alors } [\forall x \forall y (fx\delta y \leftrightarrow x=y)] \text{ sinon } \forall \delta \{ \text{suc}(\delta, \delta) \rightarrow [\forall x \forall y (fx\delta y \leftrightarrow \forall z_0 \forall z_1 \dots \forall z_{n-1} fx\delta z \wedge h_1(z_0, z_1) \wedge \dots \wedge h_k(z_{k-1}, y)] \}$

Le symbole "si a alors b sinon c" est la fonction logique de Lagrange $(a \rightarrow b) \wedge (\neg a \rightarrow c)$.

L'axiomatique minimale comprendra les axiomes suivants, en plus de la liste precedente:

- AFM(c, c^*) pour tous les symboles c de profil M

On notera F l'ensemble de ces axiomes "minimaux".

2.3.4. Realisations

Bien entendu nous appellerons realisation d'un CPPS ou CPPE tout modele, dans le CPTD associe, de l'ensemble P des axiomes minimaux.

Ayant prefere definir le formalisme avec des symboles fonctionnels, nous aurons du par souci de coherence definir directement une realisation et la valeur des formules par induction structurelle. Cela n'est possible qu'avec des ecritures trop lourdes pour etre exploitables, notamment pour definir la valeur d'un terme utilisant des fonctions partielles. Les ecritures utilisees se ramencent toutes a des consequences sur la realisation des axiomes de l'ensemble P . Il est donc plus direct de partir d'un modele de CPTD, et il n'est nullement negligeable que les ecritures en soient alors simplifiees.

Il devient alors necessaire de caracteriser les realisations ainsi obtenues.

Si nous regardons les I-types, les S-types et le N-type, nous constatons qu'ils sont independamment munis d'un systeme de symboles sans contrainte de profil. A l'exception du N-type, ils ne sont soumis a aucune contrainte d'axiome. C'est dire que l'on peut les considerer comme des calculs des predicats non types independants avec chacun sa propre realisation. La realisation du type th est ainsi une arithmetique du premier ordre, c'est-a-dire fondamentalement une arithmetique "calculable". Cela ne saurait evidemment etre une entrave.

Nous donnerons tres prochainement un exemple de ce que peuvent etre concretement de tels types.

Les axiomes $RFD(f)$ (respectivement $RID(f)$) entrainent que les realisations $\bar{f}$ des symboles sont des fonctions (respectivement des fonctions partielles).

Dans ces conditions toute composition coherente est associee a une fonction, composee des precedentes. L'axiome $AFM(f,g)$ exprime alors que $f(n,x)$ correspond a une fonction, composee n fois de g avec elle-meme. La difficulte provient du caractere partiel de ces fonctions. Definir ce qui se passe lorsqu'on utilise une fonction dans une situation non definie necessite aussi concretement le recours aux axiomes RID et AFM associes. L'ecriture $\text{mul } f \text{ } n \text{ } x$ n'avait ainsi qu'un caractere de commodite qui nous permet d'ecrire sans difficulte $\text{mul } f \text{ } n \text{ } x$ ou meme $\text{mul } \{h_k, \dots, h_1\} \text{ } n \text{ } x$. Ceci etant precise, la signification en est la meme que $f(n,x)$ et nous emploierons l'abreviation $f^n x$ ou $\{h_k \circ \dots \circ h_1\}^n x$ conformement aux habitudes mathematiques. Mais il ne s'agit que d'un abus commode.

2.3.5. Resume de la construction

Nous avons ainsi construit un formalisme derive du calcul des predicats du premier ordre avec types d'objets et avec egalite, formalisme entierement reductible au precedent, et par voie de consequence reductible au calcul des predicats non type classique.

Ce formalisme permet une expression precise des structures ou interviennent des pointeurs. Une telle construction etait necessaire pour concilier la facilite d'emploi et la definition rigoureuse des symboles associes aux fonctions partielles.

2.4. UTILISATION ET COMMENTAIRES

2.4.1.-- Completude, consistance, independance d'axiomes

Dans le traitement des questions de completude, consistance et independance des axiomes, les preoccupations concretes de l'informatique peuvent etre sensiblement differentes des preoccupations theoriques de la logique mathematique. La ou la logique a pour but de construire des fondements inattaquables aux mathematiques, nous nous interessons seulement a un outil que nous appliquons a notre cas particulier concret.

Il ne suffit pas a un informaticien que son systeme d'axiomes soit consistant. Il a besoin en plus en plus que parmi les modeles il existe des modeles finis, en un certain sens qui depend du degre de raffinement du programme. Par exemple notre calcul des predicats integre l'arithmetique, donc un modele infini. Un des pas de raffinement devra donc revenir a une situation finie, par demonstration d'un theoreme de borne ou par representation dans un modele fini.

En fait l'informaticien, ayant pour but la construction d'un algorithme realisant une representation d'une realite concrete, se donne pour but consequence de demontrer son algorithme. La consistance de son systeme est donc importante, mais prend une forme trs simplifiee d'explicitation d'un modele fini, sans utiliser l'algorithme (qui n'a pu etre valablement demontre avant d'etre sur de la consistance).

Une des caracteristiques des objets informatiques abstraits est de provenir d'un autre objet par raffinement (ou initialement par formalisation). Dans ces conditions la notion de completude n'est pas pertinente. Nous n'avons aucun besoin de completude theorique du systeme formel, mais il suffit que "la vue qu'on en a de l'exterieur" soit complete, ce qui est une notion concrete. Cette idee de "completude suffisante" exprime que les systemes formels servant a la programmation sont fondamentalement incomplets, de par leur nature de representations, et que seule doit etre complete la formalisation de l'objet initial par rapport aux operations a realiser sur lui.

La question de l'indépendance des axiomes doit être traitée dans le même esprit. Bien que deux systèmes d'axiomes soient équivalents, ils peuvent ne pas être raffinables dans les mêmes conditions. En particulier un axiome dépendant des autres peut suggérer telle représentation au détriment d'autres. Nous nous attacherons donc dans la mesure du possible à l'indépendance des axiomes, ainsi qu'à la minimalité du système.

Plus encore que l'indépendance, la minimalité du système est importante. Un axiome en trop est un choix de représentation trop précoce qui doit autant que possible être éliminé. Cette idée renforce encore la non-complétude.

En résumé un système d'axiomes intéressant pour nous a des modèles finis, est composé d'axiomes indépendants, et est minimal pour obtenir la complétude suffisante du système formel dans l'utilisation qui en est faite.

2.4.2. Un exemple d'utilisation en calcul étendu

Reprenons l'exemple de la fin du chapitre 1. Construisons un calcul des prédicats adapté à sa représentation.

- un R-type ta
- un S-type tn que nous ne distinguerons pas de th
- un symbole niv de profil $S(tn, ta)$
- trois symboles sui , une , deu de profil $C(ta, ta)$
- le symbole $<$ sera utilisé sur tn

On introduit l'abréviation suivante de profil $\langle ta, tn \rangle$:

- $L(x, n)$ signifie " $\forall m \ x = mul \ sui \ m \ niv \circ n$ "

L'axiomatique est alors la suivante:

- (a) - $\forall n \ \forall m \rightarrow \text{def } mul \ sui \ m \ niv \circ n$
- (b) - $\forall x \ \forall n \ L(x, n)$
- (c) - $\forall x \ \forall n \ L(x, n) \wedge \text{def } une \circ x \rightarrow \forall l \ l(n \wedge L(une \circ x, l))$
- (d) - $\forall x \ \forall n \ L(x, n) \wedge \text{def } deu \circ x \rightarrow \forall l \ l(n \wedge L(deu \circ x, l))$
- (e) - $\forall x \ \forall n \ \forall l \neg (L(x, n) \wedge L(x, l) \wedge n \neq l)$

où x est de type ta , n et l de type tn et m de type th .

Notre propos n'est pas pour l'instant de procéder à une étude complète de ce calcul des prédicats et de ses réalisations. Nous nous contenterons d'illustrer quelques situations classiques.

Consistance. La figure de la page suivante illustre l'existence de modèles finis de notre système d'axiomes. Souvent un tel dessin suffira pour exprimer un modèle. Les objets de type ta sont les ronds. Les flèches expriment, selon leur case de départ, la réalisation des divers symboles fonctionnels. Il n'y a donc pas lieu de refuser un tel schéma qui constitue au sens le plus strict un modèle du système d'axiomes proposé. Un tel modèle étant fini, on étudie toutes les possibilités pour calculer la valeur des axiomes.

Independance. La methode la plus frequente pour demontrer l'indepondance d'un axiome A relativement a l'ensemble des autres sera de modifier le modele pour qu'il cesse de verifier A tout en continuant a verifier les autres. Les diverses figures des pages suivantes illustrent de telles constructions.

Minimalite. La notion de minimalite est evidemment relative. Un systeme d'axiomes doit etre minimal par rapport a l'algorithme a mettre en oeuvre. Sans entrer des maintenant dans les details, nous pouvons justifier informellement la minimalite du systeme propose de cinq axiomes comme suit:

- (a), (c) et (d) sont des axiomes de finitude permettant des demonstrations d'arret des algorithmes;
- (b) est un axiome de structuration des objets de type t_a en strates liees aux objets de type t_n , et (e) est l'axiome correspondant d'unicite.

2.4.3. Introduction de proprietes du deuxieme ordre.

Il est possible de demontrer que tous les modeles du systeme ci-dessus verifient la propriete suivante:

- la relation $L(x,n)$ definit une partition des objets de type t_a qui realise une stratification du graphe sans circuit dont les arcs sont les couples $\langle x, une(x) \rangle$ et $\langle x, deu(x) \rangle$.

Des proprietes telle que celle-ci sont fondamentales pour la comprehension et l'utilisation concrete du modele, mais une telle ecriture utilise une structuration qui n'est pas definissable directement en logique du premier ordre, meme si tous les modeles finis la verifient au sens d'un autre formalisme.

Plus generalement notre calcul des predicats n'utilise comme constructeurs elementaires que des symboles correspondant a ce que notre analyse preliminaire a souligne comme elementaire: les fonctions unaires partielles. Toute construction plus complexe ne peut etre exprimee au premier ordre, ce qui exprime la necessite d'un choix de representation et du raffinement correspondant des algorithmes.

La meme necessite apparait au niveau de ce que nous avons appele CPPE (calcul des predicats du premier ordre adapte aux pointeurs, etendu) ou intervient une certaine structuration des symboles fonctionnels de profil $T(tr)$ par le biais des S-types et des symboles de profil $S(...,tr)$. Une telle structure n'est pas programmable sous cette forme. Elle necessite soit l'introduction d'une autre forme de structuration (tableau,...) associee aux langages de programmation, soit la definition d'un raffinement conduisant a un CPPS (calcul... strict) tel en plus que tous les I-types aient la representation associee finie.

Enfin il est interessant de noter des maintenant la difficulte parfois de definir les notions au premier ordre avec le minimum de symboles. Une structure aussi simple que celle de liste binaire n'est definissable qu'en introduisant la fonction reciproque des symboles fondamentaux *gch* et *dte*, ceci quel que soit le choix ulterieur de representation de la remontee (chainage de pile ou autre).

Il est donc naturel de rencontrer parfois des definitions ou des ecritures non definissables au premier ordre. Elles expriment fondamentalement la necessite d'un raffinement.

2.4.4. Le meme exemple en calcul strict.

Reprenons l'exemple precedent pour illustrer le passage d'un CPPE a un CPPS. Il nous faut représenter la fonction *niv* des entiers dans les objets de type *ta*.

Pour cela nous introduirons:

- un R-type *ts*
- deux symboles *pre* et *der* de profil *T*(*ts*)
- deux symboles *avt* et *apr* de profil *C*(*ts*,*ts*)
- le symbole *niv* est redefini de profil *C*(*ts*,*ta*)

La definition de *L* est inchangee, en prenant la variable *n* de type *ts* au lieu de *tn*. L'axiomatique est la suivante:

- (a) - $\forall n \forall m \neg \text{def mul sui } m \text{ niv}$
- (b) - $\forall x \forall n L(x,n)$
- (c') - $\forall x \forall n (L(x,n) \wedge \text{def une} \circ x) \rightarrow \forall m L(\text{une} \circ \text{mul avt } m \text{ } n)$
- (d') - $\forall x \forall n (L(x,n) \wedge \text{def deu} \circ x) \rightarrow \forall m L(\text{deu} \circ \text{mul avt } m \text{ } n)$
- (e) - $\forall x \forall n \forall l \neg (L(x,n) \wedge L(x,l) \wedge n \# l)$
- (f) - $\text{def niv} \circ \text{pre}$
- (g) - $\neg \text{def sui} \circ \text{niv} \circ \text{pre}$
- (h) - $\forall n \neg \text{def avt } n \leftrightarrow n = \text{pre}$
- (i) - $\forall n \neg \text{def apr } n \leftrightarrow n = \text{der}$
- (j) - $\forall n \text{def avt } n \rightarrow \text{apr} \circ \text{avt} \circ n = n$
- (k) - $\forall n \text{def apr } n \rightarrow \text{avt} \circ \text{apr} \circ n = n$
- (l) - $\forall n \forall m n = \text{mul apr } m \text{ pre}$

Les variables sont du type *ta* pour *x*, *ts* pour *n* et *l*, *th* pour *m*.

Cette definition est maintenant en calcul strict, c'est-a-dire intuitivement realisable sans raffinement intermediaire. Nous n'insisterons pas pour l'instant sur les questions de completude, consistance, independance et minimalite, au demeurant peu differentes du cas precedent.

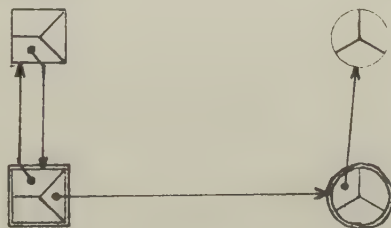
2.4.5. Modeles infinis.

Illustrons rapidement la possibilite de modeles infinis, sans entrer dans le detail de leur signification concrete.

Supprimons dans la definition precedente le symbole der et l'axiome (1) qui le reference. Ayant un modele avec k objets de type ts , nous pouvons construire un modele avec $k+1$ tels objets. C'est en entendu l'objet de l'algorithme de la section 1.4 mais on en trouve bien d'autres, par exemple l'addition d'un "monomere" selon la figure ci-dessous.

Les differents modeles Z_n ainsi construits ont la propriete d'etre contenus les uns dans les autres (y compris les fonctions en tant qu'ensembles de couples). Or nous savons qu'une chaine de fonctions a une reunion qui est encore une fonction. On peut donc considerer Z_* reunion ensembliste des Z_n qui est encore une realisation. On se convainc aisement que les axiomes sont satisfaits pour Z_* qui est donc un modele infini.

Addition d'un "monomere" (en traits gras) à un modèle existant.



2.5. RESUME ET CONCLUSION DU CHAPITRE

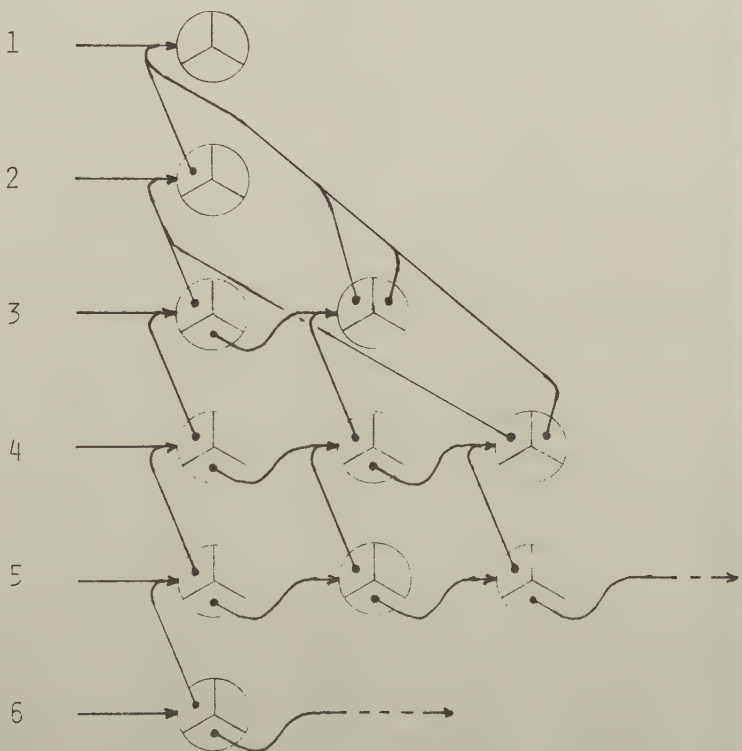
Partant d'une etude intuitive des pointeurs tels qu'ils sont utilises concretement, nous avons construit dans le formalisme du calcul des predicats du premier ordre un systeme formel. Les realisations de ce systeme formel dans sa forme stricte sont les diverses structures constructibles avec des pointeurs. Dans sa forme etendue, elles permettent d'introduire des structurations d'une autre sorte, telle que les tableaux, ou encore de definir commodement diverses structures intermediaires de raffinement.

Cette construction formelle reste cependant assez eloignee des concepts et habitudes de programmation. Elle appelle une definition plus operationnelle, quitte a revenir aux definitions les plus formelles chaque fois qu'il est necessaire de fonder avec le maximum de rigueur une construction ou une demonstration.

Modèles (emboîtés) du CPPE du paragraphe 2.3.2.

ronds : objets de type *ta*

(1) ——— aucun objet associé

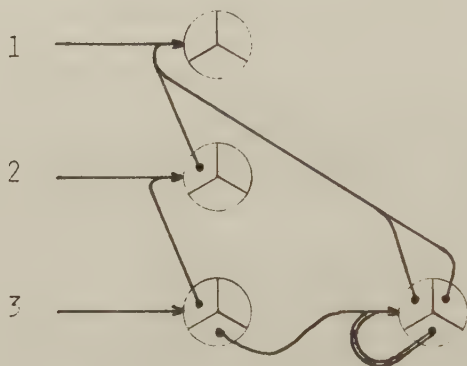


Pour chaque entier n , on a un modèle tel que $pre_n \circ niv_0 \circ p$ soit non défini pour $p > n$, et que $pre_0 \circ niv_0 \circ p$ soit défini, ce qu'on peut appeler un n -modèle.

Tout n -modèle est contenu dans au moins un $(n+1)$ -modèle.

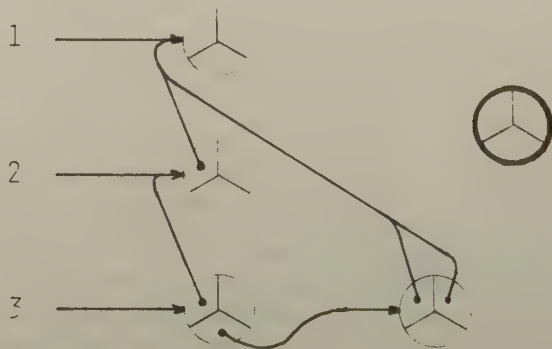
Les n -modèles sont finis, mais pour toute suite $\langle \mathcal{M}_n \rangle_{n \in \mathbb{N}}$ de n -modèles emboîtés, leur réunion est un modèle infini.

Indépendance de l'axiome (a).

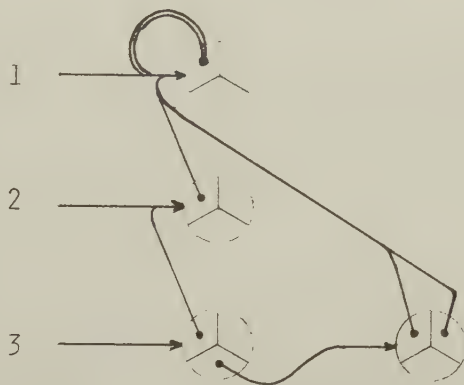


N.B. On a indiqué en trait gras ce qui est ajouté à l'objet de départ.

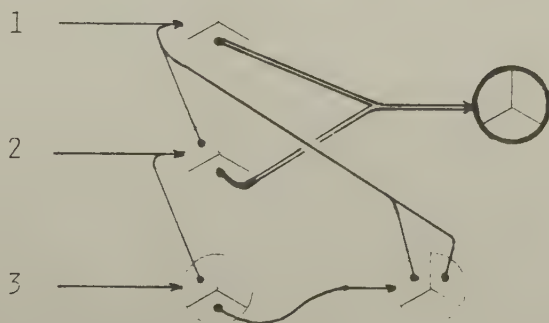
Indépendance de l'axiome (b).



Indépendance de l'axiome (c).



Indépendance de l'axiome (d).



STRUCTURES CHAINEES

3.1. IDEE DIRECTRICE.

Nous allons definir sous le nom de structures chainees un cadre general a la fois formel et proche des langages de programmation. Les objets structures associes pourront etre consideres a la fois comme realisation d'un calcul des predicats du chapitre precedent et comme etape intermediaire de la construction d'un programme, synthese formelle des divers types abstraits aisement representable au niveau d'un langage de programmation.

3.2. DEFINITION DES STRUCTURES CHAINEES.

Une structure chainee se compose d'un squelette qui permet de definir la forme des constructions realisables. Parmi ces constructions, certaines pourront etre caracterisees par le moyen d'axiomes. Enfin l'ecriture d'algorithmes transformant une construction en une autre necessitera la definition d'operateurs.

3.2.1. Squelette.

Le SQUELETTE d'une structure chainee est un triplet

$$\langle T, \langle Z_t \rangle_{t \in T}, M \rangle$$

L'ensemble T est un ensemble fini non vide. Ses elements sont appeles TYPES. Cette denomination n'appelle aucun commentaire par rapport au calcul des predicats, il est fondamentalement question du meme ensemble. Par rapport a l'utilisation informatique, a chaque type seront ulterieurement associes un ensemble de valeurs et un ensemble d'operateurs, ce qui justifie ici encore l'emploi du mot "type".

Indiquons tout de suite que nous qualifierons d'"abstrait" un ensemble pour exprimer l'idée qu'il n'est fondamentalement muni d'aucune structure. En conséquence aucune contrainte n'est mise à sa représentation, donc à la définition d'une structure particulière en facilitant l'exploitation. L'ensemble des types défini plus haut est ainsi un ensemble abstrait.

Chaque ensemble Z_t est non vide et muni d'un objet distingué z_t . On appellera les éléments de Z_t VALEURS de type T et z_t le ZERO de type T. Un ensemble de valeurs peut se réduire au seul "zero". Dans ce cas l'information qu'il porte est nulle et n'a pas besoin d'être représentée. Il revient donc au même d'autoriser des ensembles Z_t vides, mais le choix fait à l'avantage de simplifier le formalisme. Nous nous autoriserons, pour simplifier l'utilisation, l'abus de langage consistant à dire que Z_t est vide.

Aucune contrainte de finitude n'est imposée aux Z_t . Il est donc possible d'utiliser à ce niveau N, Z, R, C, ou tout monoïde libre X^* . Naturellement une telle structure ne sera pas concrètement programmable, mais un tel raffinement peut être indispensable et il ne faut donc pas contraindre la finitude à notre niveau.

Dans certains cas Z_t pourra prendre la forme d'un produit fini d'ensembles. Il n'y a donc aucune nécessité d'introduire dans le formalisme la possibilité d'un n-uplet d'ensembles de valeurs d'un type donné, ce qui compliquerait sans aucun avantage.

L'ensemble M est une fonction. Son ensemble de départ est un ensemble fini, non vide, abstrait, dont les éléments sont appelés MOULES. Son ensemble d'arrivée est l'ensemble T^2 des couples de types. En d'autres termes M est une "famille" (au sens de Bourbaki) de couples de types. On notera en surlignant $\bar{M}$ l'ensemble de départ de M.

L'ensemble des moules, préalablement défini comme abstrait, peut être structure soit lors d'un pas intermédiaire du raffinement soit par l'emploi d'un constructeur analogue à ceux employés dans les langages, en premier lieu les tableaux.

Cette formalisation à partir des moules peut sembler inutilement complexe. Elle répond à la volonté de principe de ne pas structurer a priori un ensemble sans nécessité justifiée. La formalisation habituelle utilise la notion de suite, laquelle introduit une notion d'ordre total, ou encore d'adresse, parfaitement inutile. La notion correspondante sans ordre est celle employée par Bourbaki sous le nom de "famille" [BOU58]. La situation obtenue devient intuitivement parfaitement compréhensible si on considère le moule comme nom du

couple de types et aussi comme nom de la fonction partielle qui lui sera associee. Il n'y a alors aucune raison de structurer cet ensemble de noms, sauf necessite liee au programme a realiser.

Chaque fois que cela n'entrainera aucune ambiguïte, nous autoriserons l'abus de langage consistant a appeler moule aussi bien l'objet m argument de la fonction M que le couple de types associe c ou encore l'element $\langle m, c \rangle$ de l'ensemble M . Lorsqu'il sera necessaire de preciser, nous appellerons c le PROFIL de m , en accord avec les chapitre precedent.

Les ensembles T, M sont disjoints entre eux et disjoints de chacun des Z_t , ainsi que de leurs composants dans le cas des produits.

3.2.2. Constructions

Etant donne un squelette de structure chaine, on appellera "construction" ou "objet structure" la notion correspondant a celle de realisation en calcul des predicats.

Pour une structure chaine de triplet $\langle T, \langle Z_t \rangle, M \rangle$ toute construction sera un triplet

$$\langle \langle R_t \rangle_{t \in T}, \langle I_t \rangle_{t \in T}, \langle C_m \rangle_{m \in M} \rangle$$

Les ensembles R_t sont des ensembles finis, disjoints deux a deux, disjoints des ensembles definissant le squelette. Les elements de R_t sont appeles REPERES de type t . Cette notion de repere est fondamentalement differente de toute notion d'adresse, meme si un ensemble d'adresses en machine peut etre un ensemble de reperes pour une certaine construction particuliere. Nous reviendrons sur cette question en traitant un exemple de structure chaine, en montrant que les objets fondamentaux sont effectivement les reperes.

Chaque I_t est une fonction partout definie de R_t dans Z_t appelee INFORMATION.

Pour chaque moule m , on designe par C_m une fonction partielle de R_t dans R_u ou $\langle t, u \rangle$ est le profil du moule m . Cette fonction sera appelee JONCTION de moule m . On emploiera aussi le terme de profil d'une jonction pour designer le profil du moule associe.

Il est necessaire de bien distinguer les notions ci-dessus de reperes / informations / jonctions, notions essentiellement dynamiques, des notions precedentes de types / valeurs / moules, fortement statiques meme si leur modification dans le cours d'un programme peut sembler souhaitable (idee informelle de chainage temporaire). La meme distinction se retrouve dans le programme entre les declarations de types fortement statiques (liees aux blocs) et les objets construits par le programme.

Il est également important de distinguer les notions formelles ci-dessus de leurs réalisations "canoniques" dans un langage avec pointeurs style Pascal ou PL/1. Nous continuerons à utiliser dans ce cas les noms "pointeur" et "chainage" correspondant respectivement à une valeur en un point et à la fonction réalisée.

3.2.3. Axiomatique

Le squelette d'une structure chaînée définit une structure libre. Il est nécessaire de lui adjoindre des axiomes permettant de caractériser les constructions produites par l'algorithme à réaliser ou lui servant d'argument.

Nous pouvons aisément associer à un squelette un calcul des prédicats du chapitre précédent de telle manière que toute construction du premier soit une réalisation du second. Pour cela on conserve l'ensemble des types, on traite chaque moule comme symbole de même profil, on introduit un symbole de profil $I(t)$ pour chaque type t correspondant à la fonction d'information. Les propriétés de la construction peuvent alors s'exprimer en termes de valeurs des formules dans le calcul des prédicats.

Cette façon de procéder est peu maniable. Nous suivrons donc les habitudes mathématiques en utilisant des écritures exprimant les propriétés des constructions plutôt que la valeur des formules d'un système formel.

Ceci nous impose de redéfinir un langage de formules permettant d'exprimer les axiomes sur les structures chaînées. Il ne nous semble pas utile de détailler cette définition étroitement parallèle aux précédentes. Signalons seulement les modifications:

- chaque moule est un symbole fonctionnel de même profil
- le symbole I_t est fonctionnel de profil (t)
- le type $\mathbb{N}$ est appelé "entier"
- on notera $(H_1 \circ \dots \circ H_k)^n$ au lieu de $\text{mul}(H_1, \dots, H_k) \ n \times$ conformément aux notations habituelles sur les fonctions
- on notera les connecteurs non, ou, et, $\Leftrightarrow$, $\nRightarrow$ pour les distinguer des connecteurs formels $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$.
- tous les quantificateurs sont relativisés et nous les noterons selon le type (au sens du calcul des prédicats) de la variable

$$(\exists x \mid R_t) \qquad (\exists x \mid Z_t) \qquad (\exists x \mid \mathbb{N})$$

avec leurs associés universels

$$(\forall x \mid R_t) \qquad (\forall x \mid Z_t) \qquad (\forall x \mid \mathbb{N})$$

Ceci nous fournit un langage de formules pour exprimer les axiomes de notre structure. A partir de maintenant nous appellerons STRUCTURE CHAINEE l'assemblage d'un squelette tel que defini plus haut et d'un jeu d'axiomes. En cas de besoin le squelette sans axiome pourra etre appele structure chainee libre.

Dans ces ecritures les quantificateurs relativises ne supposent pas le recours a la theorie des ensembles. C'est pourquoi nous avons employe la notation avec une barre verticale pour marquer la relativisation. On pourra aider sa propre comprehension en considerant cette barre comme un symbole d'appartenance, en se souvenant qu'il ne s'agit que d'une commodite informelle. En particulier il est rigoureusement interdit d'ecrire $(\exists n \in X)$ meme si X est une partie de N dont la fonction caracteristique est calculable.

3.2.4. References de repere, variables de repere

Pour chaque structure chainee une grande importance doit etre accordee aux couples formes d'un repere et d'un moule. Cette association d'un element dynamique (repere) et d'un element statique (moule) peut surprendre. Dans la realite elle est necessaire en tant que constituant elementaire des jonctions. La jonction associee a un moule m a un instant donne est exactement l'ensemble des couples $\langle r, C_m(r) \rangle$ ou r est un repere de type coherent dans cette situation. Fondamentalement r et m seuls definissent l'element et chaque couple ne peut etre decompose en elements plus petits. Il est donc interessant de disposer de tels constituants pour la definition des operateurs elementaires.

Nous appellerons REFERENCE DE REPERE de type u tout couple forme d'un repere de type quelconque t et d'un moule de profil $\langle t, u \rangle$.

Les "references de reperes" constituent ainsi les elements formels constitutifs des jonctions, et sur lesquels operent les operateurs elementaires. Ceci impose que tout objet ayant la meme nature soit formalise dans le meme cadre. En particulier les variables de pointeurs doivent entrer dans ce cadre. Il en resulte une definition d'apparence lourde, mais elle seule peut permettre une formalisation correcte. Inversement nous en verrons ulterieurement certains avantages:

- unicite de traitement au niveau d'un langage (helas) ideal
- bonne preparation a une representation sur disque
- facilite de construction de sous-structures.

Nous introduirons donc pour chaque type t un type annexe $t^\sim$ sans ensemble de valeurs associe, et tel que $t^\sim$ n'intervienne que dans des moules de profil $\langle t^\sim, t \rangle$. Exactement comme dans le cas general, ces moules ne sont munis d'aucune structure a priori et peuvent etre eventuellement structures au gre de l'utilisateur. Parmi les references de repere de type t se trouvent ainsi des elements de la forme $\langle v, m \rangle$ ou

est de profil $(t^{\sim}, t)$ que nous appellerons VARIABLES DE REPERE. Lorsqu'il sera necessaire de distinguer, une reference de repere qui n'est pas une variable de repere sera appelee REFERENCE PROPRE. Un moule de profil $(t^{\sim}, t)$ sera appele MOULE DE VARIABLE de type t .

La notation des references de repere sous la forme d'un couple (r, m) est lourde. Par analogie avec l'ecriture classique $rf.m$ de l'element concret associe dans les langages, nous suggerons de noter rfm cet element formel. Dans le present memoire nous le noterons rfm pour des raisons de typographie. Dans le cas des variables de repere ou un seul repere existe pour chaque type, l'indication du moule suffit pour une designation non ambigue. Nous emploierons donc l'ecriture fm (ou $!m$) pour designer la variable de repere de moule m . L'unique repere de type $t^{\sim}$ sera note w en tant que de besoin.

3.2.5. Operateurs sur les constructions

Pour toute structure chainee il existe une construction predefinie, etat de tout algorithme au debut de son execution. Cette construction vide est celle ou les ensembles de reperes sont vides pour tous les types propres. Elle contient seulement le repere particulier w permettant de definir les variables de repere. Il resulte de cette definition que les fonctions "information" et "jonction" sont vides (ensembles de couples sans element) a l'exception des moules de variables dont la jonction associee possede l'argument unique pour lequel elle n'est pas definie. Il n'existe pas d'autre construction predefinie.

Il est possible, a partir de cette construction initiale, de construire des objets structures complexes de meme squelette. Une etude preliminaire montre que quatre operateurs primitifs suffisent a definir tous les algorithmes associes aux types abstraits les plus classiques:

- creer un repere nouveau
- detruire un repere
- modifier une jonction en un point
- modifier une information en un point.

Parmi les diverses methodes de definition de ces operateurs primitifs, nous preferons la definition fonctionnelle de leur effet, plus proche a notre avis aussi bien des structures a representer que des moyens de realisation offerts par les langages. Tout operateur sera ainsi presente sous la forme

$$(R, C, I) = \text{fonc} ((S, D, J))$$

ou R et S sont les ensembles de repères, C et D les jonctions, I et J les informations.

Pour simplifier les écritures, pour tout ensemble de repères R et toute référence de repère $r = (s, m)$ nous noterons R_s la valeur pour l'argument s de la jonction associée à m . Ceci permet d'écrire sous la forme

$$R_r = \text{fct}(S_s) \quad \text{et} \quad R_p = R_p \quad \text{pour} \quad p \neq r$$

l'idée "la jonction associée au moule m est inchangée sauf au point r où elle vaut S_s , et les autres jonctions sont inchangées en tout point".

Nous pouvons maintenant définir les opérateurs.

3.2.5.1. Opérateur CREER(r)

L'argument r est une référence de repère, et soit t son type.

Soit x un objet qui n'existe ni dans M ni dans R ni dans T . On a alors les égalités suivantes qui définissent l'opérateur de création:

$$R_t = S_t \cup \{x\}$$

$$R_u = S_u \quad \text{pour} \quad u \neq t$$

$$C_r = x$$

$$C_p = D_p \quad \text{pour} \quad p \neq r$$

$$I_t(x) = z_t$$

$$I_u(y) = J_u(y) \quad \text{pour} \quad t \neq u \quad \text{ou} \quad x \neq y$$

Cette définition diffère des définitions classiques [HOA69, DEM77] en plusieurs points. Tout d'abord l'objet nouveau est introduit sans notion de suite d'objets ni d'adresse. C'est indispensable si on désire un ensemble non structure ne préjugant d'aucun choix de représentation. Cela n'entraîne aucune difficulté théorique: on obtient en théorie des ensembles fondée un objet en dehors de X en prenant le singleton $\{X\}$; si on accepte de perdre la fondation, on sait introduire des atomes [KRI69]. Il n'y a pas non plus de difficulté pratique puisque notre structure est moins contraignante. Mieux, elle autorise des possibilités nouvelles pour une organisation sur plusieurs supports, peu compatible avec les définitions classiques.

D'autres differences importantes concernent les fonctions. Notre definition impose l'"initialisation" des fonctions d'"information", en liaison avec les notions theoriques du chapitre 2 et en application de la signification de cette fonction. Cette contrainte est pour le programmeur d'ordre methodologique, mais devrait devenir une contrainte pour celui qui definit les langages ou ecrit les compilateurs.

Si nous observons la definition des jonctions, nous constatons que pour tous les moules propres les jonctions sont changees alors que l'ensemble des reperees a ete augmente. Cela signifie en termes ensemblistes que les jonctions (de profil coherent) sont non definies pour le nouveau repere. La encore il y a une contrainte de methode pour le programmeur et/ou le compilateur. Toutefois cette contrainte est ici d'une importance primordiale, en ce sens qu'elle touche la coherence meme de la structure. Si on n'"initialise" pas immediatement des la creation les "valeurs non definies" des jonctions des nouveaux objets, on obtient une entite qui n'est plus une "construction" au sens precis donne plus haut. On obtient ici n'importe quoi, alors que le cas des "informations" est moins critique.

On peut mettre cette remarque sous la forme d'un PROVERBE de la programmation avec pointeurs : toute creation d'un objet nouveau (liee aux blocs ou independante d'eux) doit etre suivie immediatement de l'initialisation de ses pointeurs a la valeur non definie. Bien entendu il serait preferable que le compilateur s'en charge.

Dans la definition de la creation, le nom r designe une reference de repere, donc un couple (repere, moule). Nous disons bien ici repere et non variable ni valeur d'une jonction. La valeur d'une jonction propre ou d'une variable ne peut etre un repere que si elle est definie. Dans toute autre situation l'operateur CREER ne serait pas defini. Traduit en termes de langages de programmation, cela nous donne un second proverbe fondamental: avant d'ecrire un dereperage ('†') il faut toujours demontrer que la variable ou l'expression qui precede est definie.

3.2.5.2. Operateur DETRUIRE(r)

L'argument r est une reference de repere, et soit t son type.

Cet operateur est defini si r est la seule reference de repere ayant sa propre valeur, notee comme precedemment S_r . On definit:

$$R_t = S_t - \{S_r\}$$

$$R_u = S_u \quad \text{pour } u \neq t$$

$$I = J \text{ restreint a } R$$

$$C_r \text{ non defini}$$

$$C_p = D_p \text{ restreint a } R \text{ pour } p \neq r$$

Cette definition est a la fois formellement non ambiguë et fonctionnellement utilisable. On peut programmer avec elle en isolant prealablement les repères a détruire dans une variable annexe ad hoc. C'est parfois lourd et inutile. Nous introduirons plus loin une notion formelle de jonction flottante qui permet de justifier l'habitude de destruction de repères non isolés du reste de la construction. Une application de cette notion permet notamment de justifier les désallocations par bloc entier employées dans les compilateurs, tout au moins lorsque la réalisation de la structure chaînée possède un minimum de propriétés supplémentaires.

3.2.5.3. Operateur VALEUR(x,v)

Les arguments sont respectivement un repère de type t et une "valeur" pour le même type.

On définit

$$R = S$$

$$C = D$$

$$I_t(x) = v$$

$$I_u(z) = J_u(z) \quad \text{pour } u \neq t \text{ ou } z \neq x$$

La remarque faite plus haut pour créer vaut ici pour valeur: l'argument est un repère et non une valeur de jonction. Il est donc nécessaire de s'assurer lorsqu'on emploie une valeur de jonction que celle-ci est bien définie.

3.2.5.4. Operateur MODIFIER(r,x)

Les deux arguments sont des références de repère du même type t .

On définit:

$$R = S$$

$$I = J$$

$$C_r = D_x$$

$$C_p = D_p \quad \text{pour } p \neq r$$

La remarque faite pour CREER et rappee pour VALEUR s'applique encore au sujet de MODIFIER. Elle concerne les deux arguments, ce qui ne met pas en cause la possibilite pour chacune de ces references d'avoir une valeur non definie.

L'operateur "modifier" decrit ici ne permet pas de remettre a "non defini" une reference de repere en dehors des cas ou x serait elle-meme non definie. Le programmeur qui a l'habitude d'ecrire ":=nil" frequemment pourrait s'en etonner. En fait nous n'avons jamais rencontre de situation concrete ou la remise a "non defini" ait un sens fondamental. Dans tous les cas il s'agit du raffinement d'une autre structuration fondamentale pour laquelle l'operateur n'existe pas ou est incorrectement realise: initialisation a la creation, adjonction / suppression de moules (defini plus loin). Nous conjecturons donc l'inutilite d'un tel operateur. S'il s'averait utile, sa definition n'offre aucune difficulte.

3.2.5.5. Sous-programmes.

Nous n'introduisons aucun operateur particulier de traitement des sous-programmes. Nous supposons seulement que peuvent etre transmis comme arguments en mode constante et comme resultats de fonctions aussi bien les references de repere que les moules ou les valeurs. Nous avons ainsi en particulier tout ce qui est necessaire pour considerer comme sous-programmes coherents nos quatre operateurs primitifs.

3.3. DEFINITION EN STRUCTURE CHAINEE DE L'EXEMPLE DE 1.4

Reprenons l'exemple defini en 1.4 et complete plus loin en 2.3.2 a 2.3.4. Nous etions parvenus a un algorithme exprime en termes de listes et a une definition de la structure en termes de notre calcul des predicats adapte aux pointeurs. Nous pouvons maintenant definir structure et algorithme en termes de structures chainees, sous la forme ou elles doivent etre concretement utilisees. La presente section doit ainsi etre comprise comme la poursuite du traitement complet, du probleme au programme, de l'exemple. Nous repartons bien entendu de ce qui a deja ete elabore.

3.3.1. Definition de la structure.

Deux types A (pour arborescence) et S (pour strate).

Deux moules de variables PRE et DER de type S et les divers moules de variables qui seront temporairement necessaires dans le cours de l'algorithme.

Moules propres :

- AVT et APR de profil (S,S)
- NIV de profil (S,A)
- SUI,UNE,DEU de profil (A,A)

Axiomes (rappel) ecrits formellement sous la forme indiquee avec les structures chainees. On donne la definition du symbole abreviatif L dans lequel x est un repere de type A et n de type S.

$$L(x,n) \Leftrightarrow (\exists m!N) \text{ sui}^m \circ \text{pre}(n) = x$$

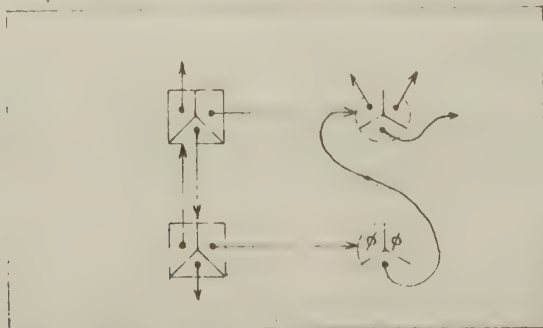
On a les douze axiomes :

- (a) $(\forall n!S)(\exists m!N) \text{ sui}^m \circ \text{pre}(n)$ non defini
- (b) $(\forall x!A)(\exists n!S) L(x,n)$
- (c) $(\forall x!A)(\forall n!S) (L(x,n) \text{ et } \text{une}(x) \text{ defini}) \Leftrightarrow$
 $(\exists m!N) L(\text{une}(x), \text{avt}^m(n))$
- (d) $(\forall x!A)(\forall n!S) (L(x,n) \text{ et } \text{deu}(x) \text{ defini}) \Leftrightarrow$
 $(\exists m!N) L(\text{deu}(x), \text{avt}^m(n))$
- (e) $(\forall x!A)(\forall n, l!S) \text{ non } (L(x,n) \text{ et } L(x,l) \text{ et } n \neq l)$
- (f) nivpre defini
- (g) $\text{sui} \circ \text{niv} \circ \text{pre}$ non defini
- (h) $(\forall n!S) \text{ avt}(n)$ non defini $\Leftrightarrow n = \text{pre}$
- (i) $(\forall n!S) \text{ apr}(n)$ non defini $\Leftrightarrow n = \text{der}$
- (j) $(\forall n!S) \text{ avt}(n) \text{ defini} \Leftrightarrow \text{apr} \circ \text{avt}(n) = n$
- (k) $(\forall n!S) \text{ apr}(n) \text{ defini} \Leftrightarrow \text{avt} \circ \text{apr}(n) = n$
- (l) $(\forall n!S)(\exists m!N) n = \text{apr}^m \circ \text{pre}$

La meme axiomatique peut etre rendue plus intuitive et plus proche des raffinements anterieurs comme suit: (a) pour chaque strate on a une liste lineaire simple d'arborescences (b) chaque arborescence est dans une de ces listes ... (e) ... et une seule (c,d) pour UNE et DEU on accede a la liste d'une strate inferieure (f,g) la liste associee a PRE a une arborescence et une seule (h,i,j,k,l) les strates sont organisees en liste lineaire double.

Nous donnons les indications fondamentales permettant de demontrer l'independance des principaux axiomes (pour la liste double, voir le chapitre 5). Il n'est de toute maniere pas concretement indispensable pour la programmation de realiser ces demonstrations. Leur role est essentiellement theorique. Dans tous les cas ici nous modifions une construction satisfaisant les axiomes pour en obtenir une qui ne satisfait plus un et un seul des douze axiomes.

- (a) on fait bouclier tout ou partie des arborescences queues de listes sur elles-mêmes via SUI
- (b) on introduit une arborescence isolée
- (c) pour tout ou partie des x tels que $UNE(x)$ soit définie, on établit $UNE(x)=x$
- (d) analogue à (c) en employant DEU
- (e) on introduit une arborescence sans descendance suivant la figure qui suit



- (f) on introduit une strate vide
- (g) on duplique l'arborescence sans descendance (à cause de (c) et (d)) dans la liste associée à PRE.
- pour les axiomes de liste double, l'indépendance sera étudiée ultérieurement au chapitre 5.

Nous avons déjà schématisé (figure 2.2) Une construction correspondant à une partie de la preuve de consistance. Il ne semble pas utile de surcharger l'exposé en complétant cette preuve.

3.3.2. Raffinement de l'algorithme

Le programmeur traitant cet exemple est censé avoir précédemment étudié les structures fondamentales, notamment tout ce qui concerne la liste linéaire. Cependant nous préférons ici illustrer des maintenant notre définition des structures chaînées, même si certains développements doivent être différés.

Supposant donc connues les listes linéaires, nous obtenons l'algorithme suivant:

```

:  algorithme unsommetdeplus (n : entierpositif)
:  creer(fv)
:  modifier(vfavt,fder)
:  modifier(derfapr,fv)
:  modifier(fder,fv)
:  modifier(thm,derfavt)
:  modifier(fa,hmfniv)
:  tant que a defini faire
:      ajouter(fa,fvide)
:      modifier(fa,afsvi)
:      fin tant que
:  modifier(tbm,tpre)
:  pour m entre 1 compris et n div 2 exclu faire
:      modifier(thm,hmfavt)
:      modifier(fa,hmfniv)
:      tant que a defini faire
:          modifier(fb,bmfniv)
:          tant que b defini faire
:              ajouter (fa,fb)
:              modifier(fb,bfsvi)
:              fin tant que
:          modifier(fa,afsvi)
:          fin tant que
:      modifier(tbm,bmfapr)
:      fin pour
:  si n pair alors
:      modifier(fa,hmfavtfniv)
:      tant que a defini faire
:          modifier(fa,fb)
:          tant que b defini faire
:              ajouter(fa,fb)
:              modifier(fb,bfsvi)
:              fin tant que
:          modifier(fa,afsvi)
:          fin tant que
:      fin si
:  fin algorithme
:
:
:  sous-algorithme ajouter(a,b:reference de repere de type arborescence)
:  creer(tp)
:  modifier(ptune,a)
:  modifier(pdeu,b)
:  si d defini alors modifier(dtsui,tp)
:      sinon modifier(derfpre,tp)
:      fin si
:  modifier(td,tp)
:  fin sous-algorithme

```

Nous avons dans cette écriture utilise "HM↑AVT↑NIV" qui n'est pas encore defini. Rappelons que "XXX↑ZZZ" est une autre écriture pour la reference de repere "(XXX,ZZZ)". Nous definirons par extension "VVV↑XXX↑ZZZ" comme equivalent a "TTT↑ZZZ" dans lequel TTT est la valeur courante de la reference de repere "VVV↑XXX". Cette definition s'etend recursivement a toute écriture complexe "...↑...↑...↑...↑..." Par remplacement de VVV.

3.3.3. Demonstration de cet algorithme

Nous n'allons pas donner ici une demonstration mathematique complete et formalisee de cet algorithme. Notre but d'ensemble est concret: il s'agit de mettre en lumiere les fondements, outils et methodes particulieres de la programmation avec pointeurs. Nous estimons donc plus utile d'ecrire la demonstration telle qu'un programmeur devrait la faire. Il faut d'ailleurs remarquer que les mathematiciens eux-memes operent d'une maniere tres abreegee lors de leurs demonstrations. Ce n'est qu'en cas de difficulte particuliere ou de desaccord qu'ils ont recours aux fondements logiques, a l'exception de l'etablissement des quelques theoremes les plus fondamentaux. Nous ne saurions donc ni innover ni perdre de la rigueur par cette pratique.

Toutes les variables locales A, B, D, BM, HM, V sont censees etre non definies a l'entree de l'algorithme.

Commencons la demonstration. Tout d'abord occupons nous de la definition des reperes r dans les references de reperes rfc. Dans tous les cas le repere est porte par une variable. Les variables V et P sont toujours definies comme resultats de creations. Les variables A, B, D sont toujours definies aux moments de leurs utilisations puisqu'elles sont protegees derriere un test de cette contrainte. Pour BM et HM on commence par remarquer que l'algorithme augmente exactement d'une unite le nombre d'elements de la liste des strates. Il y a donc exactement N strates lors de l'appel avec argument N. Pour toute valeur de M, les variables BM et HM contiennent le M-ieme element des listes de strates respectivement directe et inverse. Sachant que M est inferieur a N/2, on en deduit que HM et BM sont toujours definies.

On verifie que les sequences d'insertion en liste correspondent bien a ce qui a ete demontre lors de l'etude a priori des listes lineaires. Precisons seulement que D est la queue de la liste simple definie par DERTNIV et SUI. Ceci justifie notamment notre affirmation sur le nombre d'elements de la liste des strates.

De la remarque donnee plus haut sur BM et HM il resulte que dans l'iteration controlee par M la variable A enumere la liste L(N-M) et B la liste L(M). De meme la premiere iteration controlee par A enumere L(N) et la derniere enumere L(N/2) pour N pair.

Toutes les iterations intervenant dans l'algorithme s'arretent en les considerant isolement. En particulier l'algorithme entier s'arrete.

Les autres proprietes utiles de l'algorithme viennent des pas precedents du raffinement, et notamment l'unicite des arborescences dans l'enumeration, ainsi que les axiomes (c) et (d).

3.4. COMPLEMENTS DIVERS.

3.4.1. Problemes de destruction - jonctions "flottantes"

3.4.1.1. Divers choix possibles de destruction.

Lors de la definition de l'operateur DETUIRE, nous avons signale la necessite d'approfondir son etude. C'est en effet l'operateur non seulement le plus mal defini dans les langages, mais aussi le plus difficile a concevoir.

Nous avons choisi de definir la destruction dans le cas ou le repere a detruire possede une reference et une seule. C'est assurement le plus facile sur le plan theorique, mais le programmeur est alors oblige d'assurer cette unicite, soit en la demontrant soit en programmant la conservation d'un compte de references sur chaque objet.

Il est parfois facile d'obtenir a partir des axiomes une evaluation du compte de references en fonction des diverses mesures de l'objet a representer. Ce sera le cas dans le projet evoque au chapitre 7. Par contre cela peut etre notablement plus delicat, comme dans l'exemple qui nous a servi jusqu'ici. On peut difficilement envisager une solution universelle. Il faudra etudier la destruction des structures classiques et indiquer les grandes lignes de la resolution generale, qui laissera une part importante de demonstration particuliere.

Il n'est guere plus facile d'organiser un compte de references. Aucun langage a notre connaissance ne le realise actuellement, notamment pour des raisons d'efficacite. Or aucun langage ne permet une realisation valable des operateurs primitifs, faute de notion associee a celle de moule et qui soit utilisable comme argument de procedure. Il faudra donc traiter les operateurs comme des macro-instructions et les recopier a la main.

Aucune de ces deux solutions n'est pleinement satisfaisante. Selon le probleme effectif l'une ou l'autre sera plus aisee, parfois meme l'une ou l'autre en fonction des diverses sous-structures a detruire.

Si la realisation du langage de programmation a choisi d'incorporer un compte de references, un autre choix pour l'operateur de destruction devient possible: on detruit tout repere dont le compte de references est devenu nul. Outre les defauts generaux du comptage des references, cette solution n'est pas theoriquement satisfaisante. Elle cache la destruction alors que le programmeur doit savoir a quel moment precis son compte devient nul pour chaque repere. Le programme est moins aise a comprendre et a verifier. Nous pensons que toute destruction doit etre explicitement demandee. Par ailleurs cette solution cache une des difficultes des structures chainees: la question fondamentale n'est pas la nullite du compte de references d'un repere, mais son accessibilite a partir des variables. La destruction se ramene alors a un algorithme de ramasse-miettes ("garbage collector") lequel est hors de propos au niveau de raffinement ou nous nous placons. Nous rejetons donc totalement cette deuxieme solution.

Une troisieme solution pour la destruction est definie par les ecritures suivantes, avec les memes notations que la definition donnee initialement.

$$R_t = S_t - \{S_r\}$$

$$R_u = S_u \quad \text{pour } u \neq t$$

$$I = J \text{ restreint a } R$$

$$C = D \cap R^2$$

autrement dit cet operateur detruit en meme temps le repere et toutes ses references quelles qu'elles soient. Il n'y a pas ici de difficulte theorique, mais c'est pratiquement irrealisable sous cette forme. La manipulation des fonctions inverses des jonctions n'est pas envisageable des que la structure n'est plus elementaire. On peut s'en convaincre en observant seulement notre exemple, pourtant simple. Nous appellerons DESTRUCTION FORTE cet operateur.

3.4.1.2. Notion de reference flottante.

Une extension assez simple de notre formalisme permet de rendre concretement utilisable cet operateur. Nous integrons a toute construction un ensemble F initialement vide. La contrainte d'emploi de l'operateur initial de destruction est supprimee et on ajoute a la definition une nouvelle ligne

$$F_{\text{final}} = F_{\text{initial}} \cup \{S_r\}$$

Ce nouvel operateur est maintenant partout defini, il n'a aucun effet si r est non defini. On l'appellera DESTRUCTION FAIBLE.

On introduit maintenant une relation d'equivalence sur toute construction, qui est la plus fine rendant equivalents entre eux et equivalents a la valeur non definie tous les elements de F. En d'autres termes plusieurs representations sont possibles pour la valeur non definie.

Nous appellerons REFERENCES FLOTTANTES les references de repere ayant pour valeur un element de F.

Etudions maintenant le quotient des constructions par l'equivalence ci-dessus. La creation est inchangee sous reserve de choisir le nouveau repere hors de F. La modification est inchangee. Le test des valeurs non definies doit evidemment tenir compte de F. Le quotient de la destruction faible est la destruction forte: toutes les references au repere detruit sont elles-memes detruites.

Ainsi par une extension suivie d'un quotient on ouvre la voie a la realisation de l'operateur de destruction sous sa deuxieme forme. Nous appellerons cette methode: METHODE DES JONCTIONS FLOTTANTES (ou des references flottantes). Il reste a voir comment la mettre en oeuvre.

3.4.1.3. Mise en oeuvre concrete.

En resume il est donc possible de laisser "flotter" certaines references de repere apres destruction du repere qui etait leur valeur. Le resultat est correct a condition de savoir reconnaitre comme "non definies" de telles references.

Il ne nous semble pas envisageable de realiser effectivement une telle structure-quotient. L'usage que nous preconisons de la methode est le suivant. On cherche a detruire une structure complexe en evitant au maximum la reduction du compte de references. On commence par elaborer une enumeration des reperes a detruire. La contrainte est un peu plus stricte que la simple "non repetition". L'algorithme d'enumeration ne doit jamais reutiliser un repere prealablement rencontre dans l'enumeration. On peut alors detruire "faiblement" les reperes dans l'ordre de l'enumeration. Certaines references deviennent "flottantes" jusqu'a destruction du repere qui les porte.

A la fin d'une telle enumeration il ne doit plus rester de references flottantes. Les references flottantes ne doivent donc etre portees que par des reperes promis a la destruction. Pour tout autre cas une reduction prealable du compte de references est necessaire.

Cette methode des jonctions flottantes n'est pas une veritable construction theorique nouvelle. Elle n'est que l'adaptation de la theorie principale a la solution simplifiee d'un probleme complexe par nature. Il suffit donc de noter que cette application est justifiee et surtout dans quelles conditions elle est applicable.

3.4.2. Application : detruire les tables d'un compilateur.

Nous allons considerer un exemple non trivial d'application des structures chainees: les tables d'un compilateur pour un langage a structure de blocs. Nous allons montrer comment les problemes de destruction dans une telle structure s'integrent a notre formalisme, malgre les apparences.

Pour un langage a structure de blocs, les tables du compilateur constituent une structure chainee que nous qualifierons de stratifiee. Chaque repere a un certain niveau, lequel est un entier naturel, et la structure verifie la condition:

pour tout repere x , toute jonction f , on a $\text{niv}(x) \{ \text{niv}(f(x))$.

le probleme pose est de detruire tous les reperes du niveau superieur. On sait donc en particulier qu'aucun repere a conserver ne reference un repere a detruire et qu'il n'y a donc aucun besoin de reduction prealable du compte de references.

Supposons au depart que nous disposions d'une enumeration n'utilisant aucun moule lie aux tables elles-memes. On peut par exemple imaginer disposer d'une liste lineaire des reperes du niveau a detruire, mais d'autres solutions simples existent. Nous nous trouvons ainsi dans les conditions d'applications de la methode des jonctions flottantes.

Nous avons ainsi reduit notre probleme initial. Comment pouvons-nous maintenant enumerer les reperes du niveau a detruire? Le choix que nous ferons provisoirement est de realiser ce balayage comme operateur primitif sur les tables, et donc de le programmer au cours du raffinement suivant.

Jusqu'ici tout est independant du langage et de sa mise en oeuvre. Nous introduisons maintenant une contrainte qui n'a aucune raison d'etre universelle, mais qui existe dans Pascal [HOA73] et dans les formalisations habituelles [DEM77] des pointeurs. Nous supposons que la chronologie des creations de reperes induit sur les reperes une relation d'ordre total accessible au programmeur. Dans ces conditions on peut ecrire le compilateur de telle maniere que les reperes pour chaque niveau constituent un intervalle de cet ordre total. L'ordre sur les intervalles est en meme temps isomorphe a l'ordre sur les niveaux.

Sous ces diverses contraintes, les repères à détruire sont donc caractérisés par une relation de la forme $r \succ r$, ce qui fournit une première possibilité d'énumération.

On simplifie une dernière fois le programme en remarquant qu'avec la méthode des jonctions flottantes, les destructions de repères d'un même niveau sont deux à deux permutable. Il est donc possible de les réaliser en parallèle, ce qui se raffine aisément en introduisant un opérateur de destruction de tous les repères supérieurs à un certain repère donné.

Ainsi notre formalisme est parfaitement compatible avec les habitudes de compilation. Il leur apporte ici l'avantage d'une meilleure compréhension des conditions d'emploi de telle ou telle structure classique. De manière générale il sera souvent intéressant de considérer comme primitifs certains opérateurs sur les structures chaînées pour pouvoir ensuite les réaliser avec le maximum d'efficacité. Cela pourra en particulier servir à définir des algorithmes de contrôle ou de reconstitution d'intégrité pour des constructions permanentes sur support insuffisamment fiable (disque).

3.4.3. Opérateurs de modification des structures.

Jusqu'ici nous avons considéré le squelette de la structure invariant tout au long du programme. En réalité il n'est pas concevable de conserver une telle contrainte. Tout programmeur connaît le besoin de variables locales pour la réalisation d'un algorithme. Parfois un algorithme nécessite un moule temporaire pour conserver certaines informations. Par ailleurs on sait ajouter et retirer des types en liaison avec les entrées et sorties des blocs. Il nous faut donc introduire des opérateurs supplémentaires agissant sur le squelette. Il s'agit là bel et bien d'opérateurs qu'il faudra ensuite raffiner explicitement.

3.4.3.1. Ajouter un type, ajouter un moule.

Ces deux opérateurs d'addition n'entraînent aucune difficulté. Toute construction associée à un squelette demeure une construction pour tout autre squelette obtenu par additions à partir du premier.

3.4.3.2. Suppression d'un moule de variable

La suppression la plus simple est celle d'un moule de variable. Elle correspond à la notion bien connue de destruction des variables locales à la sortie de leur bloc de définition. Toute construction se réduit immédiatement à une construction pour le squelette réduit. Il est seulement nécessaire de s'assurer que tous les repères demeurent accessibles dans la construction réduite, et éventuellement d'assurer des destructions préalables.

Toutefois la comparaison avec le mouvement des blocs ne doit pas être poussée trop loin. Nous n'imposons aucune contrainte théorique aux additions et suppressions des moules de variables. Nous pensons en particulier que l'écriture correcte d'un algorithme exige de donner aux noms de variables la durée d'existence minimale, le compilateur se chargeant du travail d'allocation lorsqu'il est possible statiquement, ou générant le code qui lui correspond. C'est donc une question de mise en œuvre que nous examinerons dans le chapitre 4. On peut cependant remarquer dès maintenant que l'on peut intuitivement représenter les additions / suppressions des moules de variables par des créations / destructions et que la mise en œuvre semble donc possible.

3.4.3.3. Suppression d'un moule propre

Rappelons que nous distinguons comme moules propres ceux qui ne sont pas des moules de variables. Un tel opérateur sera valide si la construction argument a une des formes particulières lui permettant d'être aussi une construction pour le squelette obtenu après la suppression, et si tout repère y demeure accessible. Il n'y a alors aucune opération physique sur la construction. On peut envisager en dehors de ce cas de valider un opérateur de réduction des constructions. Formellement l'opérateur de suppression du moule m consistera en la suppression de la jonction associée C_m , possible dans tous les cas. Il demeure deux difficultés majeures: la mise en œuvre, que nous verrons au chapitre 4, et la démonstration d'accessibilité.

3.4.3.4. Suppression d'un type

Il n'y a pas de difficulté formelle: en supprimant le type du squelette il faut supprimer tous les moules ayant ce type dans leur profil. Dans les constructions il faut alors détruire tous les repères de ce type ainsi que les jonctions associées aux moules supprimés. Cet opérateur est très différent de la simple sortie d'un bloc ou un type est défini (cas des langages du style Pascal). En effet, sans addition de moules, toute jonction référencera des repères dont le type est dans un bloc englobant. La méthode des jonctions flottantes exposée plus haut suffit à réaliser ce cas, au demeurant analogue au problème de destruction dans les compilateurs. Des problèmes nouveaux de mise en œuvre apparaissent dès qu'on accepte d'ajouter des moules. On peut alors vouloir ajouter un moule (t,u) où le bloc de déclaration de t englobe strictement celui de u . Nous y reviendrons dans le chapitre sur les mises en œuvre.

REALISATION DES STRUCTURES CHAINEES

4.1. CADRE DU TRAVAIL

Ayant defini dans le chapitre precedent un type abstrait, nous allons maintenant consacrer un chapitre a sa realisation concrete dans divers langages. Nous donnerons a cette etude un caractere fondamentalement pratique. En particulier ce ne sera pas notre propos d'envisager la situation de la multitude de langages existants, meme au travers de leur classification en groupes coherents. Il nous semble preferable de choisir un tres petit nombre de langages en fonction de leur utilisation reelle pour resoudre les problemes de la classe qui justifie l'emploi des structures chainees.

Nous nous interesserons donc a Fortran-IV et a PL/1, les deux principaux langages industriels. Nous ne nous occuperons donc pas de Cobol. Les applications de gestion qui utilisent ce langage ne sont qu'indirectement de la classe "ensembliste" qui nous interesse. L'emploi de structures chainees en Cobol correspondra presque toujours au traitement d'une structure permanente sur disque. Ce type de situation constitue comme nous l'avons deja signale un sujet d'etude en soi, dont nous dirons quelques mots.

Les problemes prennent en effet une forme tres differente selon qu'on traite un algorithme en memoire centrale detruite en fin de calcul, ou bien que l'on travaille a modifier une structure preexistante sur disque, a conserver ensuite. L'integrite de la structure entraine des contraintes severes sur la realisation des operateurs primitifs. Ce cas doit etre etudie separement.

Nous nous interesserons a Pascal, moins pour son utilisation effective que par son caractere de langage minimal permettant la programmation structuree jusqu'au bout. Egalement parce que c'est devenu par la meme la base obligatoire de tout nouveau langage, dont on constate un debut d'extension industrielle.

Nos remarques sur les langages a tableaux s'applique aux langages d'assemblage, et surtout aux langages structures de meme niveau appeles couramment PL ou LP. Notre interet pour eux n'est pas fortuit. En effet nous verrons dans le courant du chapitre les difficultes profondes a traiter certains operateurs du langage de haut niveau. Le recours a des langages de bas niveau est donc parfois obligatoire, bien que la solution reellement structuree et efficace consiste dans l'utilisation de langages a plusieurs niveaux. Dans ce dernier cas certains operateurs devront etre ecrits en langage de bas niveau. Dans tous les cas ce type de langage merite d'etre envisage.

Nous prendrons ces divers langages tels qu'ils sont. Nous ne pensons pas justifie de definir un langage specifique pour les structures chainees. Nous pensons d'abord qu'il est possible de programmer correctement dans les langages existants, meme si certains algorithmes doivent etre amenes. Ensuite l'important parait etre de definir pour tous les langages quelles sont les structures disponibles minimales d'abord et optimales ensuite pour realiser les structures chainees dans les meilleures conditions. Cette definition fera l'objet du chapitre 6, le present chapitre prenant le caractere explicite d'etude de situations existantes.

4.1.1. STRUCTURE DE BASE

Commencons par donner une premiere idee intuitive des rapports entre les notions formelles et les objets d'un langage de programmation ou d'une machine. Un examen critique nous approchera ensuite de la realisation.

STRUCTURE CHAINEE	LANGAGE	MACHINE
type	type (en gros)	-
=====	=====	=====
moule	champ	adresse
	d'article	relative fixe
=====	=====	=====
reference	champ ou	adresse d'un
de repere	variable	mot contenant
	par reference	une adresse
=====	=====	=====
variable	variable	mot contenant
de repere		une adresse
=====	=====	=====
modifier	affectation	affectation
ou valeur		
=====	=====	=====
creer/dispose	new/dispose	a realiser
	allocate/free	
=====	=====	=====
repere	a la fois l'	a la fois une
	article et	zone et son
	son pointeur	adresse

Pour la plupart des notions formelles une notion machine simple et naturelle existe. Cette situation est assez previsible dans la mesure ou la notion de pointeur (informellement) est deja une abstraction de certains aspects des machines. Nous avons conu le formalisme pour en eliminer toute notion d'adresse et meme toute notion d'ordre sur les repere (explicite dans [HOA69] et dans [DEM77]).

Il est souhaitable de l'eliminer au maximum dans les langages evolues. Au nive au de la machine, il convient au contraire de la rehabiliter. La memoire convenablement interpretee est fondamentalement une construction (au sens precis que nous avons donne a ce mot). Dans ces conditions la notion associee au repere est tout autant la zone allouee que son adresse, mais il faut la distinguer soigneusement de son contenu comme on distingue habituellement le nom de la valeur a laquelle on l'a affecte.

Si on accepte cette notion d'adresse pour realiser la notion de repere, l'affectation est maintenant tres proche de l'operateur "modifier". Dans de nombreux cas l'approximation est suffisante. Dans quelques cas pourtant la finesse de representation impose un raffinement supplementaire de l'affectation, nous y reviendrons.

Le principal travail de representation a realiser concerne le couple creer/destruire a traiter comme un tout. Aucun langage ne le traite correctement aujourd'hui. Compte tenu de la contradiction habituelle entre securite et efficacite le traitement devrait etre laisse a la discretion du programmeur parmi un choix propose a lui. Il faut bien en conclure que les operateurs de creation/destruction devront toujours etre realises explicitement, c'est une securite indispensable.

4.1.2. Dans des langages avec pointeurs.

Dans les langages disposant de pointeurs (Pascal, PL/1) il existe des operateurs de bibliotheque pour la creation et la destruction. Ces operateurs provoquent, le premier une allocation de place de taille convenable et le retour de l'adresse, le second renvoie la valeur "nil" et recupere parfois l'adresse. Par rapport au formalisme ce choix de destruction est coherent. A l'oppose l'allocation n'est jamais totalement satisfaisante, puisque l'initialisation n'est jamais realisee. Donc il sera toujours indispensable de realiser un operateur de creation, ce qui tend a imposer la definition d'une procedure homologue de destruction qui peut en definitive de reduire sans perte de securite a une remise a "nil" de son argument.

Cette solution simple peut etre demontree a partir de la definition axiomatique pour Pascal [HDA73]. En fait elle se demontre a partir du seul fait que la zone nouvelle est disjointe de toutes les zones couramment allouees, ce qui est un minimum qu'on peut exiger de tout langage.

Des difficultes apparaissent dans les programmes qui detruisent frequemment. Deux solutions existent alors pour la recuperation de la place. La meilleure serait sans doute de fournir au programmeur les moyens d'ecrire lui-meme (en assembleur ou langage structure de bas niveau) sa propre bibliotheque d'execution, ou simplement de lui fournir differents choix, de securite et rapidite differentes. La

solution par défaut est de gérer soi-même une organisation des repères détruits pour les recréer en priorité.

Il y a dans ce cas introduction d'un niveau de raffinement supplémentaire de gestion des repères détruits. Formellement cela revient à introduire deux moules supplémentaires de liste (premier, suivant) à ce niveau, avec remplacement des anciens axiomes par des nouveaux ou les quantifications de repères sont relativisées au complémentaire de la liste d'objets "libres". On peut alors créer soit par une création réelle soit par retrait d'un élément "libre", la destruction correspondant à un ajout comme élément "libre". On démontre dans le formalisme lui-même que la sous-construction des éléments "occupés" est isomorphe à la construction correspondante de l'algorithme. La réalisation concrète d'une telle structure ne présente aucune difficulté majeure (voir l'annexe, page 3 du programme). Les éléments "libres" sont par simplicité gérés en pile. Souvent il est commode d'ajouter une organisation des repères "occupés" (liste linéaire double) notamment pour permettre une sauvegarde de la construction par balayage des repères "occupés".

Dans une structure chaînée à un seul type, cette solution est simple et efficace. Lorsque plusieurs types coexistent, il faut choisir entre une gestion unique de zones dont le type dépend des conditions d'accès et une gestion séparée des différents types. Le premier cas est typiquement celui des langages non types, style PL/1. Dans les langages fortement types, style Pascal, l'emploi des articles avec variante est une solution tentante. En fait elle entraîne la perte du contrôle statique des types, contrainte difficilement supportable. On pourrait améliorer avec un contrôle dynamique [NGU78] ou mieux avec un contrôle de type plus fin (voir section 6.2).

4.1.3. Représentation avec des tableaux.

Dans les langages sans pointeurs, on sait parfaitement mettre en œuvre une représentation des structures chaînées. L'objet fondamental associé à la notion de repère est un indice pour un certain tableau, tableau d'articles dans le meilleur des cas, tableau à deux dimensions à défaut d'article. Le système d'allocation / désallocation doit alors toujours être réalisé, et les principes évoqués restent valables. On obtient alors une représentation totalement non typée, source d'erreurs bien connue.

Il n'y a pas de différence fondamentale avec les langages à pointeurs explicites. La notion de référence de repère est maintenant obtenue avec un indice d'une part et d'autre part soit un nom de champ soit un indice sur la seconde dimension. Il y a donc toujours moyen de réécrire "à la main" un algorithme en structures chaînées dans un tel langage. Mais peut-on trouver une solution permettant de traiter rapidement

```

        modifier((repere1,moule1),(repere2,moule2))
par quelque chose comme
        moule1(repere1) := moule2(repere2)
ce qui faciliterait la comprehension et la reecriture?

```

On s'aperçoit rapidement de la difficulté de cette entreprise. Parmi les langages évoqués seul PL/1 semblerait convenir avec la construction "ISUB". Par exemple on pourrait écrire les déclarations correspondant à 3.2.1 comme suit:

```

:      dcl CONSTR ( 3 , MAXSTR ) fixed ;
:      dcl ( AVT , SUI ) ( MAXSTR ) syn CONSTR ( 1 , isub ) ;
:      dcl ( APR , UNE ) ( MAXSTR ) syn CONSTR ( 2 , isub ) ;
:      dcl ( NIV , DEU ) ( MAXSTR ) syn CONSTR ( 3 , isub ) ;
:      dcl INDXALLOC fixed initial ( 0 ) ;

```

Cette solution conceptuellement satisfaisante et raisonnablement efficace a le lourd défaut de ne pas être universellement acceptée (cf. PL/1 pour IRIS-80).

Une solution intermédiaire consisterait à écrire la même instruction sous la forme:

```

CONSTR ( MOULE1 , REPERE1 ) := CONSTR ( MOULE2 , REPERE2 )

```

déjà bien moins lisible. On s'aperçoit rapidement que cela n'a de valeur réelle qu'à condition de disposer de constantes (pour l'indice associé aux moules) définies pour toute la durée du programme conçu obligatoirement comme un tout indivisible. Cela élimine tout espoir avec Fortran.

On constate en définitive une profonde inadaptation de ces langages à l'emploi des tableaux pour réaliser des structures chaînées. Un préprocesseur spécialisé serait ici la meilleure solution.

4.1.4. Réalisation sur disque.

Une troisième grande catégorie de réalisations des structures chaînées utilise une simulation de mémoire permanente. La taille et/ou la permanence de la construction exige l'emploi d'une mémoire de masse, disons pour fixer les idées d'un fichier sur disque avec une méthode d'accès simulant une mémoire de grande capacité.

Une telle structure a exactement les propriétés d'un tableau dont les éléments sont les informations immédiatement associées à un repère, c'est-à-dire soit un article soit un tableau linéaire de bornes fixes. La différence avec la représentation correspondante en mémoire principale tient à l'impossibilité d'accès individuel aux composants d'article. On dispose uniquement de deux opérateurs primitifs

- ACCES (indice , article)
- RANGE (indice , article)

qui assurent un transfert effectif entre la memoire permanente et la memoire centrale. Outre cette difference fondamentale il faudra tenir compte du cout en temps de ces operateurs.

Tout algorithme ecrit a partir du formalisme des structures chainees convient a une representation sur disque, des qu'on a realise les operateurs primitifs. Mais une telle representation a des caracteristiques tres particulieres qui necessitent son etude specifique:

- la question de place disponible est d'importance tres diminuee
 - l'optimisation des acces-disque doit etre prise en consideration
 - l'integrite de la construction doit etre garantie ou recalculable.
- Les contraintes d'integrite devront porter sur l'algorithme et peu sur la representation, les contraintes de temps ont l'aspect inverse, la place ayant une situation intermediaire.

Le principal interet d'une representation sur disque est sa permanence. La construction existante doit donc garantir a son utilisateur une axiomatique fixee au depart, ceci quelles que soient les conditions de l'arret du precedent algorithme actif. Il est hors de question de reactiver l'algorithme de creation. Ceci ne depend pas seulement de la qualite de l'algorithme mais aussi de l'etat du systeme d'exploitation. En cas d'arret intempestif on est ainsi dans une situation caracteristique simple: de deux acces-disque consecutifs, le premier est correct mais non le second, l'algorithme s'etant arrete a ce moment (entre les acces ou acces incorrect). S'il n'est pas toujours possible d'assurer totalement l'integrite, il est possible en general d'offrir une securite raisonnable des que la structure chainee verifie une condition minimale: il doit etre possible a partir de tout repere de connaitre de maniere immediate les structures auxquelles il appartient. Pour tout element de liste il faut par exemple connaitre la tete de liste ou toute information caracteristique de cette tete. Un programme approprie est ainsi capable de verifier et de retablir (a cout en general eleve) l'integrite de la construction. Definir ces moyens de verification et de reconstitution fait partie de la definition d'une structure destinee a etre representee sur disque. C'est en general facilite par l'absence de contrainte de place pour les representations, qui autorise l'addition d'd'informations immediates.

4.1.4.1. Representation non optimisee.

La representation des moules variables, simplifiee dans la memoire centrale, necessite d'etre explicite sur memoire permanente. Il faut ainsi un repere special en exemplaire unique, origine des jonctions associees aux moules de variables. L'accès a la structure se fait a partir de cet article qu'il est donc necessaire de caracteriser (l'adresse fixe 0 ou 1 convient fort bien, mais toute autre caracterisation simple est acceptable). Nous insistons sur le fait que la necessite de cet "article de tete" ne provient pas d'un choix de

programmation mais de la formalisation des moules de variables, ce qui tend a les justifier a posteriori.

Cette representation des variables etant donnee, on doit ecrire l'algorithme en formalisant correctement les variables et la realisation des operateurs primitifs ne presente aucune difficulte particuliere. Les operateurs d'allocation / liberation sont ici donnees en supposant l'existence d'une pile d'elements libres. Les variables T et V designent des zones en memoire centrale permettant l'acces a la memoire permanente, autrement dit ce sont des seconds arguments possibles pour ACCES et RANGER.

```

:  operateur MODIFIER ((r,m),(s,n))
:  ACCES (r,T)
:  v := T.m
:  ACCES (s,T)
:  T.n := v
:  RANGE (s,T)
:  fin
:
:
:  operateur CREER ((r,m),type)
:  ALLOCATION (v,type)
:  ACCES (r,T)
:  T.m := v
:  RANGE (r,T)
:  fin
:
:
:  operateur DETRUIRE ((r,m),type)
:  ACCES (r,T)
:  v := T.m
:  T.m := representation de "nil"
:  RANGE (r,T)
:  LIBERATION (v,type)
:  fin
:
:
:  operateur ALLOCATION (w, )
:  ACCES(variable,V)
:  w := V.premier libre { depend eventuellement du type u }
:  si w non defini alors sortie en erreur
:  ACCES (w,T)
:  V.premier libre := T.libre suivant
:  RANGE (variable,V)
:  { initialisation de T faite dans LIBERATION et INITIAL sauf
:  eventuellement le type u et la place de "libre suivant" }
:

```

```

: RANGE (w,T)
: fin
:
:
:
:
: operateur LIBERATION (w, )
: ACCES (w,T)
: "initialisation" de T { remises a "nil" }
: ACCES (variable,V)
: T.libre suivant := V.premier libre
: V.premier libre := w { depend eventuellement de u }
: RANGE (w,T)
: RANGE (variable,V)
: fin
:
:
:
:
: operateur INITIAL (variable,maximal)
: initialisation d'article de tete dans V
: "initialisation" de T { comme dans LIBERATION }
: v := maximum
: tant que v } variable faire
:   RANGE (v,T)
:   T.libre suivant := v
:   v := v-1
:   fin tant que
: V.premier libre := variable + 1
: RANGE (variable,V)
: fin

```

Ainsi on peut aisement realiser une structure chainee sur disque avec une quantite d'accès assez lourde. On compte en effet 3 mouvements (2 lectures, 1 écriture) pour modifier et 6 mouvements (3 dans chaque sens) pour créer ou détruire. Cela se réalise avec deux articles d'échange T et V, ou mieux dans un langage type avec un article par type de la structure. On obtient ainsi une réalisation simple, aisement démontrable, mais peu efficace.

4.1.4.2. Représentation totalement optimisée.

Face à l'inefficacité de la solution précédente, une solution optimisant les accès est possible. Le moyen est de conserver en mémoire centrale tous les articles lus, associés à leur indice en mémoire permanente. On introduit ainsi un niveau logique d'accès. Par exemple on utiliserait un court tableau d'articles $T[*]$ et on écrirait

```

:  operateur MODIFIER ((r,m),(s,n))
:  ACCES LOGIQUE (r,i)
:  ACCES LOGIQUE (s,j)
:  TC[j].n := TC[i].m
:  RANGE LOGIQUE (s,j)
:  fin

```

On a alors besoin d'une table des repères ayant une copie en mémoire, avec comme information l'indice associé dans T, et un booléen d'égalité entre les articles homologues en mémoire et sur disque.

```

:  operateur ACCES LOGIQUE (r,i)
:  ENTABLE (r,i,ok) { localisation de r dans la table }
:  si non ok alors      { cas où l'article est déjà accessible }
:  ALLOUER (r,i,ok)
:  si non ok alors      { cas où il faut une libération préalable }
:  CHOIX (i)    { l'indice le plus anciennement référence }
:  si non EGALITEC[i] alors RANGE ( REPEREC[i],TC[i] ) finsi
:  finsi
:  ADRESSEC[i] := r
:  EGALITEC[i] := vrai
:  ACCES (r,TC[i])
:  finsi
:  fin
:
:
:  operateur RANGE LOGIQUE (s,j)
:  { assertion d'entrée : REPEREC[j] = s }
:  EGALITEC[j] := faux
:  { en variante de sécurité on ajouterait
:  RANGE (s,TC[j]) }
:  fin

```

Si on ne sait rien sur l'algorithme on obtient par cette méthode une excellente optimisation des accès avec une taille réduite pour T, L'augmentation de cette taille alourdit la charge de recherche (ENTABLE, ALLOUER) sans avantage notable. La valeur optimale semble être le nombre maximal de variables de repère utilisées simultanément par un algorithme (par exemple 14 pour le programme du chapitre 7).

Cette réalisation approche ou assure l'optimum des accès au disque. Elle manque malheureusement de sécurité. La présence en mémoire centrale de nombreuses informations non recopiées rend difficile la reconstitution en cas d'erreur. On perd de nombreux accès en rangeant systématiquement après chaque opération (la "variante de sécurité" de l'opérateur ci-dessus). Il est alors commode, efficace et sûr de ranger systématiquement l'article "de têtes" et tous les articles n'ayant plus aucune référence par une variable de l'algorithme.

4.1.4.3. Autres choix de representations.

Si nous partons d'une representation comme ci-dessus avec un article d'accès (les "T" ci-dessus) par variable de l'algorithme, il devient interessant de proceder par transformation du programme pour adapter la representation a l'algorithme. Les transformations interessantes sont celles qui aboutissent aux resultats suivants:

- elimination d'appels de ACCESLOGIQUE lorsque son assertion de sortie est verifiee sans appel
- calcul statique de EGALITE et simplification des tests correspondants
- suppression des recherches pour ALLOUER par calcul statique de valeurs convenables, puis suppression de CHOIX.

Ces transformations sont de nature semantique, ce qui elimine l'espoir d'une solution simple et generale a ce probleme de simplification d'algorithme. Inversement la plupart des types abstraits de base (listes de natures diverses et leurs cas particuliers) sont associes couramment a des simplifications quasi-canoniques.

4.1.4.4. Conclusion sur les representations permanentes.

Les cas concrets les plus frequents et les tailles les plus importantes sont evidemment associes a des structures construites dans une memoire a caractere permanent. La complexite des structures y limite fortement la possibilite d'utilisation des types abstraits inclus dans les langages, sans indication explicite de representation. C'est donc un domaine qui ne peut se passer des pointeurs. Nous pouvons constater avec satisfaction que notre formalisation, ecrite pour les realisations en memoire centrale, non seulement s'applique aisement mais en plus qu'il fournit une base feconde a l'elaboration de representations efficaces.

Cependant les representations permanentes continuent a poser des problemes tres specifiques qui constituent un sujet d'etude en eux-memes, independamment de la notion de structure chainees.

4.2. OPERATEURS SUR LA STRUCTURE

La section precedente etudie la realisation des structures chainees et des operateurs sur les constructions dans trois groupes de situations recouvrant les cas concrets rencontres effectivement. Il faut maintenant completer par la realisation des operateurs sur le squelette de la structure.

4.2.1. Addition et suppression de moules.

La definition formelle prevoit la possibilite en cours de programme d'ajouter ou de supprimer un moule, aussi bien un moule propre qu'un moule de variable. En ce qui concerne les moules propres cette possibilite ouvre la voie a l'utilisation de "chainages temporaires" pour la duree d'un algorithme. C'est de toute facon un evenement assez rare.

Il est evidemment possible dans tous les cas de definir le ou les moules temporaires comme moules permanents, ceci a des changements de nom pres. De la meme facon des moules permanents peuvent porter des informations de nature differente pourvu que leurs portees soient disjointes. On peut donc toujours transformer un programme avec addition et suppression de moules (propres ou non) en un programme n'utilisant pas ces operateurs.

On se passe difficilement de tels operateurs des qu'il faut inserer ou enlever des algorithmes. Si ces operateurs existent, le texte de l'algorithme se suffit a lui-meme. Dans le cas contraire il faut realiser des modifications dans les declarations du programme appelant, ceci a la main avec tous les risques d'erreur qui en resultent.

Ces deux operateurs ne sont applicables que sous contrainte severe: on ne peut ajouter qu'un moule qui n'existe pas et on ne peut supprimer qu'un moule existant. En outre il ne semble pas tres contraignant d'imposer que le controle soit purement statique, ce qui est possible en liant la portee des moules temporaires aux procedures. Dans ces conditions la notion de moule temporaire n'est qu'un probleme de compilation pour les representations en memoire centrale (voir chapitre 6).

La situation est un peu differente pour les representations permanentes. Les contraintes d'integrite redeviennent primordiales. Les principes generaux restant les memes, certaines utilisations courantes en memoire centrale doivent etre interdites des que l'on detruit la possibilite de reconstruction, meme si cette destruction n'est que temporaire.

L'aspect le plus delicat des additions/suppressions de moules propres est la necessite d'initialisation. Tout moule cree est associe a une jonction definie nulle part, donc dans la representation a un champ "nil" pour tous les reperes existants au moment de la creation du moule. La situation ideale laisse ce travail a la bibliotheque d'execution, sinon il est necessaire de le realiser par le programme de raffinement soit des additions soit des suppressions. Dans certains cas la methode des references flottantes permettra des simplifications aussi de ce travail.

4.2.2. Addition et suppression de moules de variables.

On peut naturellement appliquer le cas general des moules quelconques aux moules de variables. Cependant les utilisations et proprietes differentes justifient d'envisager des traitements separees. Comme pour les moules propres c'est une aide importante a la conception et a la demonstration de definir pour les moules de variables une portee minimale. Une programmation structuree correcte devrait pouvoir disposer d'un outillage fin en ce sens. Contrairement aux moules propres nous avons une utilisation frequente de tels operateurs. Il est donc necessaire de definir un moyen effectif de declarer une variable pour la duree d'une instruction, ce qui est satisfaisant pour le controle statique des contraintes d'unicite. Pour la compilation de telles declarations on remarque les proprietes suivantes:

- la sequence "creer X; detruire X" est sans effet
- la sequence "detruire X; creer X" a un effet isomorphe a la remise a "nil" de la variable X (que nous noterons NX pour simplifier)
- "creer X" et "detruire X" permutent avec tout operateur ne faisant pas intervenir X.

Bien entendu dans tous les cas il est faux d'affirmer l'identite des constructions obtenues en appliquant les deux suites de fonctions considerees. On peut seulement affirmer qu'elles sont indiscernables pour tout algorithme sur la structure chainee, et il suffit pour cela qu'elles soient isomorphes pour le squelette. Sur la base de ces trois proprietes on demontre les trois equivalences de programmes suivantes (on simplifie "creer X" en CX et "detruire X" en DX):

- si I ne contient pas X alors I equivaut a "CX;I;DX"
- "tantque B faire CX;I;DX fait" equivaut a
"tantque B faire CX;I;DX fait; CX;DX" lui-meme equivalent a
"CX; tantque B faire I;DX;CX fait; DX" ou encore a
"CX; tantque B faire I;NX fait; DX"

*

- "selon V parmi [k₁:debut CX;I₁;DX fin] finselon" equivaut a

"CX;selon V parmi [k₁:I₁]* finselon; DX"

On peut ainsi pour tout programme satisfaisant la contrainte statique d'unicite, calculer statiquement un programme equivalent ou les ajouts et suppressions de variables sont reportes au debut et a la fin de la procedure (cette contrainte entraine notamment qu'il n'y a pas d'occurrence de X dans B et V).

Dans tous les langages actuels, avec ou sans outil de definition explicite de pointeurs, l'utilisation des operateurs d'addition et suppression d'un moule de variable est donc seulement une commodite de raffinement, facilitant ecriture et demonstration.

Dans une representation permanente, le raffinement de ces operateurs peut entrainer une programmation explicite dans les cas ou on a optimise les acces-disque. La notion de creation d'une variable peut en effet entrainer un besoin explicite d'allocation dans une table, la destruction entrainant a l'inverse la liberation associee et le mouvement des comptes de references. Ces effets dependant fortement de la representation choisie, nous ne les detaillerons pas plus. Il faut par contre noter que les isomorphismes de structures donnees plus haut ne concernent que la structure chainees et que les realisations n'ont aucune raison de demeurer isomorphes.

4.2.3. Ajout et suppression de types.

Ajouter un type revient a creer un ensemble vide de reperes de ce type et cette operation est en general suivie d'ajout de moules. Inversement supprimer un type est une operation bien plus complexe de destruction d'ensembles de reperes accompagnee de la suppression de tous les moules ayant ce type comme destination. On retrouve donc des operateurs deja connus et etudies plus haut, avec une organisation des elements "libres" du type argument qui doit imperativement etre definie dans le programme principal.

4.3. REMARQUES GENERALES SUR LES REPRESENTATIONS

L'expose de ces quelques methodes de representation des structures chainees recouvre les cas essentiels rencontres explicitement en programmation. Bien entendu il ne visait pas a etre exhaustif. En particulier une structure chainee peut etre representee en utilisant pour partie chacune des situations. Elle peut aussi avec le meme formalisme etre repartie entre plusieurs machines. D'autres organisations des elements "libres" et "occupes" sont possibles.

Nous avons montre dans ce chapitre la possibilite de maniere simple de représenter toute construction pour tout squelette de structure chainee. Tous les operateurs sont realisables, aussi bien ceux qui modifient la construction que ceux qui modifient le squelette. Il est ainsi possible de programmer correctement et en surete dans les langages les plus divers, mais nous avons constate la necessite permanente de reecritures a la main liee a l'inadaptation des langages. Nous etudierons au chapitre 6 diverses possibilites d'amelioration des langages existants.

STRUCTURES DE BASE CLASSIQUES

5.1. LISTE LINEAIRE: DEFINITION AXIOMATIQUE

5.1.1. Systeme formel de base.

Nous appellerons structure de liste binaire toute structure definie par le systeme formel ci-dessous ou qui en derive directement par extension.

Le systeme formel "liste lineaire" est defini par:

- un ensemble annexe E
- l'ensemble S des objets du systeme
- un symbole fonctionnel $vi : \rightarrow S$
- deux symboles fonctionnels $aj, su : E \times S \rightarrow S$
- les axiomes donnees plus loin

Dans l'etude d'un tel systeme formel interviennent des expressions parfois complexes utilisant les symboles fonctionnels de base (les "termes" du systeme). Dans une situation comme la notre l'ensemble des objets du systeme n'apparait jamais plus d'une fois dans le profil (au sens mathematique du mot) des symboles fonctionnels. Nous simplifierons alors les termes en n'utilisant pas des fonctions aj et su de $E \times S$ dans S, mais les familles de fonctions de S dans S obtenues par abstraction du second element:

$$\begin{aligned} aj_e &= \lambda s. aj\ e\ s & aj_e(s) &= aj(e, s) \\ su_e &= \lambda s. su\ e\ s & su_e(s) &= su(e, s) \end{aligned}$$

Nous etendrons cette notation aux compositions de telles fonctions, et nous introduirons la notation aj_A lorsque A est une suite d'elements de E avec la definition

$$aj_{e_1, \dots, e_n} = aj_{e_1} \circ \dots \circ aj_{e_n}$$

Nous conservons l'ordre de la suite pour eviter d'introduire trop de complications, malgre l'emploi de la composition de fonctions avec son sens d'evaluation de droite a gauche.

Ce systeme formel a trois axiomes, exprimes ci-dessous sous la forme d'egalites entre fonctions.

$$(AS1) - su_e v_i = v_i$$

$$(AS2) - su_e aj_e = id_S$$

$$(AS3) - su_e af_f = aj_f su_e \quad \text{pour } e \neq f$$

5.1.2. Forme normale.

Nous savons, en supposant le systeme consistant, qu'on obtient un modele en prenant l'ensemble des expressions coherentes ecrites avec su , aj (e parcourant E) et vi , puis en prenant le quotient de cet ensemble par l'equivalence la plus petite contenant les axiomes. Il est interessant de pouvoir definir dans chacune des classes d'equivalence un element distingue (appele alors "forme normale") et un algorithme permettant de le calculer pour toute expression. C'est ce que nous allons faire en commençant par un petit lemme technique qui simplifiera les demonstrations.

Lemme. Soit A une suite d'elements de E , ne contenant pas e . Alors $su_e aj_A vi = aj_A vi$. Soit A une suite d'elements de E contenant e . Ecrivons $A = B ! (e) ! C$ ou B ne contient pas e . Alors $su_e aj_A vi = aj_B ! C vi$.

Demonstration. D'apres l'axiome $AS3$, su_e permute avec tous les aj_f pour $f \neq e$. Ainsi $su_e aj_A vi = aj_A su_e vi$ dans le premier cas et $su_e aj_A vi = aj_B su_e aj_e aj_C vi$ dans le second. On conclut immediatement par $AS1$ dans le premier cas et $AS2$ dans le second.

Notons des maintenant que tous les axiomes ont ete utilises par la demonstration.

Corollaire. Pour toute suite A d'elements de E et tout element e de E , on a $su_e aj_A vi = aj_B vi$ ou B est une sous-suite de A .

Theoreme. Toute expression obtenue par composition de fonctions aj_e et su_e , et appliquee a vi , est egale a une expression de la forme $aj_A vi$ ou A est une suite d'elements de E .

Demonstration par recurrence sur le nombre des occurrences des fonctions su_e dans l'expression. Si ce nombre est nul, on est deja dans la situation cherchee. Sinon on isole la premiere occurrence (gauche) d'un su_e : $f = aj_f su_e h$ et on applique a h l'hypothese de recurrence. On a alors $f = aj_f su_e aj_A vi$ auquel on applique le lemme, ce qui donne $f = aj_f aj_B vi$.

Algorithmiquement on obtient le resultat en isolant cette fois la derniere (a droite) occurrence d'un su_e , soit $f = g su_e aj_A vi$. On lui applique le lemme pour obtenir $f = g aj_B vi$ qui a une occurrence de su_e en moins. Ce processus est itere et donne en un nombre fini de pas une expression $f = aj_C vi$.

Le parallele entre les formes normales obtenues ci-dessus et les suites d'elements de E peut se poursuivre:

Theoreme. Considerons le monoide libre E^* construit sur E , muni des operations:

- vi est le mot vide
- aj_e est la concatenation a gauche de la suite reduite a $\langle e \rangle$
- su_e est la suppression de la premiere occurrence a gauche de e

On obtient ainsi un modele de "liste lineaire".

On verifie immediatement les trois axiomes.

Corollaire. Le modele defini plus haut est isomorphe a E^* muni des trois operations qui precedent. En particulier, la forme normale $aj_A vi$ est unique.

Demonstration. Tout modele pouvant etre obtenu par quotient a partir du modele canonique, la fonction $aj_A vi \rightarrow A$ etant un morphisme, il en resulte l'impossibilite de $aj_A vi = aj_B vi$ si A est different de B , d'ou les deux resultats.

Cette caracterisation du modele canonique fournit une bonne base intuitive pour la comprehension des listes lineaires. Donnons immediatement un resultat fondamental complementaire.

Theoreme. $su_e su_f = su_e su_f$.

Demonstration. On peut se contenter de demontrer le resultat lorsque l'argument des fonctions est une forme normale $aj_A vi$. On procede par recurrence sur la longueur de la suite A . Pour une suite $A = \langle g \rangle$ ou $B = \langle g \rangle$ on distingue deux cas selon que g est e (resp. f) ou n'est ni l'un ni l'autre.

- $su_f su_e aj_e aj_B vi = su_f aj_B vi = su_e aj_e su_f aj_B vi = \dots$
 $\dots = su_e su_f aj_e aj_B vi$
- $su_e su_f aj_g aj_B vi = aj_g su_e su_f aj_B vi = \dots$
 $\dots$ auquel on applique l'hypothese de recurrence

5.1.3. Liste lineaire ordonnee.

Supposons maintenant l'ensemble annexe E muni d'un ordre total (note $\succ$). On introduit deux nouveaux symboles fonctionnels

$$- a_e, s_e : E \times S \rightarrow S$$

que nous utiliserons sous la forme

$$- a_e, s_e : S \rightarrow S$$

Ces symboles sont soumis aux axiomes suivants:

$$(A01Z) \quad a_e v_i = a_e v_i$$

$$(A01S) \quad a_e a_f = a_f a_e \quad \text{pour } e \succ f$$

$$(A01E) \quad a_e a_e = a_e$$

$$(A01I) \quad a_e a_f = a_e a_f \quad \text{pour } e \prec f$$

$$(A02Z) \quad s_e v_i = v_i$$

$$(A02S) \quad s_e a_f = a_f s_e \quad \text{pour } e \succ f$$

$$(A02E) \quad s_e a_f = \text{identite de } S$$

$$(A02I) \quad s_e a_f = a_f \quad \text{pour } e \prec f$$

Ces axiomes sont donnees sous une forme permettant le calcul immediat de la valeur de a_e ou s_e en cas d'application a une forme normale. On peut en deduire assez simplement l'existence d'une forme normale $a_A v_i$ pour les expressions en a_e , s_e et v_i , mais ce resultat ne caracterise pas suffisamment la forme normale obtenue.

5.1.4. Forme normale des listes ordonnees.

Nous procederons ici encore par lemme technique preliminaire.

Lemme 1. Soit A une suite strictement croissante d'elements de E et soit e un element de E. Alors $a_e a_A v_i = a_D v_i$ ou D est une suite strictement croissante d'elements de E.

Lemme 2. Dans les memes conditions $s_e a_A v_i = a_D v_i$ ou D est encore une suite strictement croissante.

Demonstration. Il est commode pour la demonstration de privilegier l'element initial d de la suite A, puis de proceder par cas selon la comparaison de d et e.

- si A est vide $ao_e aj_A vi = ao_e vi = aj_e vi$ (A01Z)
- si A est vide $so_e aj_A vi = so_e vi = vi$ (A02Z)
- si $d > e$ $ao_e aj_A vi = aj_e aj_A vi$ (A01I)
- si $d > e$ $so_e aj_A vi = aj_A vi$ (A02I)
- si $d = e$ $ao_e aj_A vi = aj_A vi$ (A01E)
- si $d = e$, $A = \langle e \rangle ! B$ alors $so_e aj_A vi = so_e aj_e aj_B vi = \dots$
 $\dots = aj_B vi$ (A02E)

- si $d < e$, on peut decomposer $A=B!C$ ou les elements de B sont strictement inferieurs a e, et ceux de C superieurs ou egaux. On a alors par (A01S) et (A02S) $ao_e aj_A vi = aj_B ao_e aj_C vi$ et $so_e aj_A vi = aj_B so_e aj_C vi$ ce qui nous ramene a l'un des cas precedents.

A partir de ces deux lemmes on peut reduire toute expression en ao_e et so_e appliquee a $aj_A vi$ (ou A est strictement croissante) en une expression de meme forme mais avec une occurrence de ao ou so en moins (cette occurrence etant la derniere). En iterant le processus on obtient le theoreme:

Theoreme. Toute expression en ao_e et so_e appliquee a vi possede une forme normale $aj_A vi$ dans laquelle A est une suite strictement croissante d'elements de E.

La caracterisation sera complete avec le resultat suivant:

Theoreme. Soit A une suite strictement croissante d'elements de E. Alors $aj_A vi = ao_A vi$.

Demonstration par recurrence sur la longueur de A. Pour la longueur zero, c'est $vi=vi$ trivial. Pour la longueur un, c'est l'axiome (A01Z). Si c'est vrai pour la longueur n non nulle, soit $B=\langle e \rangle ! A$ une suite strictement croissante de longueur n+1. On a alors:

$$\begin{aligned}
 ao_B vi &= ao_e ao_A vi && \text{(definition de la notation } ao_B) \\
 &= ao_e aj_A vi && \text{(hypothese de recurrence)} \\
 &= aj_e aj_A vi && \text{(axiome A01I)} \\
 &= aj_B vi && \text{(definition de la notation } aj_B)
 \end{aligned}$$

Corollaire. Toute expression en ao_e et so_e , appliquee a vi , possede une forme normale $ao_A vi$ dans laquelle A est une suite strictement ordonnee d'elements de E.

5.1.5. Proprietes complementaires des listes lineaires ordonnees.

Ainsi les expressions realisables avec ao_e , so_e et vi constituent une partie R de S, bien entendu stable pour ao_e et so_e .

Dans l'isomorphisme defini plus haut avec E^* , R est en bijection avec les mots strictement croissants. Les operations ao_e et so_e sont alors associees aux suivantes:

- ao_e : ajout de e s'il n'existe pas encore a l'unique place conservant l'ordre; non-operation si e est deja dans la liste

- so_e : suppression de l'unique occurrence eventuelle de e

Ces operateurs isomorphes nous permettent d'etablir sans difficulte les resultats suivants.

Theoreme. Soit R l'ensemble des expressions $ao_A vi$ ou A est une suite quelconque d'elements de E. On a les proprietes suivantes (quantifiees relativement a R):

$$(\forall r \in R) \quad so_e so_e r = so_e r$$

$$(\forall r \in R) \quad so_e r = so_e r$$

$$(\forall r \in R) \quad ao_e ao_e r = ao_e r$$

$$(\forall r \in R) \quad ao_e ao_f r = ao_f ao_e r$$

$$so_e vi = vi$$

$$(\forall r \in R) \quad so_e ao_e r = r$$

$$(\forall r \in R) \quad so_e ao_f r = ao_f so_e r \quad \text{pour } e \neq f$$

En particulier les axiomes (AS1), (AS2) et (AS3) sont verifiees par les restrictions a R de ao_e et so_e . On en deduit immediatement

$$(\forall r \in R) \quad so_e so_f r = so_f so_e r$$

Corollaire. Soient A et B deux suites d'elements de E telles que l'ensemble des elements de A et celui de B soient identiques. Alors $ao_A vi = ao_B vi$.

Ce dernier resultat nous donne enfin le resultat de caracterisation comparable a celui sur les listes lineaires avec E^* .

Theoreme. Le modele canonique des listes lineaires ordonnees sur E est isomorphe a L'ensemble des parties finies de E muni de:

- vi : l'ensemble vide
- aj_e : ajout du singleton $\{e\}$
- sq_e : retrait du singleton $\{e\}$

5.1.6. Independance des axiomes.

Nous allons nous preoccuper de l'ind independance des axiomes de definition de la liste lineaire simple, que nous rappelons:

- (AS1) $su_e vi = vi$
- (AS2) $su_e aj_e = \text{identite}$
- (AS3) $su_e aj_f = aj_f su_e$ pour $e \neq f$

Dans les trois cas, partant d'un modele de ces trois axiomes, nous modifions les operateurs pour fournir un modele de deux seulement des axiomes mais ne satisfaisant pas le troisieme.

Independance de (AS1). Definissons $aj'_e = aj_e$, $su'_e = su_e$ et $vi = aj_A vi$ ou A est une suite quelconque non vide. On a toujours (AS1) et (AS2) mais pas (AS3). En prenant un A convenable on peut remplacer (AS1) par des expressions assez variees. Remarquons qu'avec (AS1) et (AS2) seuls on trouve une forme normale $aj_B su_C vi$ ou B et C sont disjoints et B contenu dans A .

Independance de (AS2). Definissons $aj''_e = aj_e$ pour tout e ou partie (non vide) des elements e de E . On conserve $su''_e = su_e$ et $vi'' = vi$. Alors (AS1) et (AS3) sont conserves mais pas (AS2).

Independance de (AS3). La question n'a pas de sens si E n'a qu'un element. Si E a au moins trois elements, soit e un element d'ordre trois d'une permutation p de E . On definit $vi''' = vi$, $aj'''_e = aj_{p(e)} aj_e$ et $su'''_e = su_e su_{p(e)}$ de maniere a conserver (AS1) et (AS2). Soient alors $x = aj'''_e su'''_e vi'''$ et $y = su'''_{p(e)} aj'''_e vi'''$. On a:

- $x = aj_{p(e)} aj_e su_{p(e)} su_{p(p(e))} vi = aj_{p(e)} aj_e vi$
- $y = su_{p(e)} su_{p(p(e))} aj_{p(e)} aj_e vi = aj_e vi$

On a alors $\text{su}_e y = vi$ et $\text{su}_e x = aj_p(e) vi$ donc $x \# y$. Malheureusement cette methode ne permet pas de conclure si E n'a que deux elements.

5.2. LISTE LINEAIRE : DEFINITION FONCTIONNELLE

La definition axiomatique qui precede rencontre rapidement des limites a son utilisation. Les notions de sous-liste ou d'insertion en un endroit determine n'y sont pas definissables. Il manque pour cela a cette formalisation une notion informatique classique, celle de "place", quelle qu'en soit d'ailleurs la nature.

En introduisant cette notion, nous creons en meme temps la distinction (naturelle a notre avis) entre l'objet et l'information qu'il porte. La definition axiomatique ne traite que l'information, ce qui constitue une difficulte fondamentale par rapport a notre definition des pointeurs: rupture logique de la sequentialite physique des donnees.

5.2.1. Definitions generales et exemples.

On appellera structure de liste lineaire simple sur un ensemble E la donnee de deux fonctions partielles

- une constante "premier" de E
 - une fonction "suivant" de E dans E
- telles que soient verifiees les deux axiomes suivants
- il existe un element x de E tel que "suivant" soit non defini en x , ou bien E est vide
 - tout element de E peut etre obtenu en appliquant la fonction "premier" puis un nombre suffisant de fois "suivant".

Cette definition s'applique tres generalement a de nombreux objets reels, tant qu'on ne pose pas le probleme de creer de tels objets. Ceci exprime l'idee fondamentale que de nombreux objets informatiques permettent un "balayage" sous une forme normalisee:

```

:   X := premier
:   tant que X est defini faire
:     travail sur X { ne modifiant pas X }
:     X := suivant(X)
:   fin tant que

```

La notion de liste est en ce sens une necessite liee a la nature sequentielle des machines, aussi bien de leur fonctionnement que pour leurs donnees.

On trouve en particulier comme exemples de liste lineaire simple:

- les tableaux a une dimension (avec seulement une partie de leurs operateurs)
- les listes a pointeurs (que nous developperons plus loin)
- les fichiers sequentiels (la variable de balayage est alors cachee)
- tout segment terminal d'une autre liste lineaire
- toute sous-liste definie par un critere de choix
- toute arborescence munie de fonctions "premier prefixe" et "suivant prefixe" (resp. infixe ou postfixe)
- ainsi fondamentalement que tout ensemble totalement ordonne, des qu'on lui a associe un ensemble convenable de reperes.

La difference d'essence entre ces exemples apparaissent a la creation de tels objets complexes. Nous reserverons alors le nom de liste lineaire simple aux structures permettant la construction avec les operateurs primitifs suivants:

- ajouter X derriere Y defini
- ajouter X en tete
- retirer derriere Y defini
- retirer en tete.

En d'autres termes nous nous interesserons aux listes lineaires utilisant un mecanisme de pointeurs, ce mot etant pris au sens informel du chapitre 1. Il est donc raisonnable de partir de la definition des listes en termes de structures chainees.

5.2.2. Definition comme structure chainee.

Nous appellerons finalement liste lineaire simple la structure chainee definie par le squelette:

- un type unique (qu'en consequence nous ne nommons pas)
- un moule de variable "pre"
- un moule propre "sui"
- un ensemble d'informations quelconque

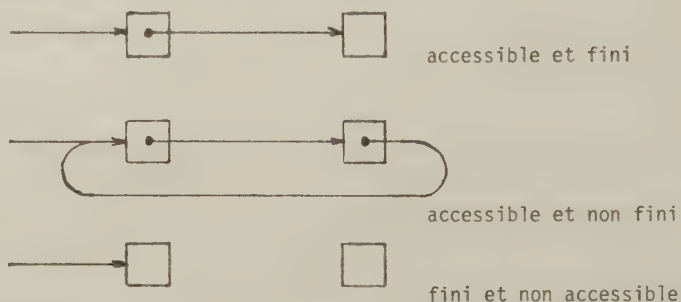
et les axiomes:

(LS1) $(\forall x) (\exists n!N) \text{ sui}^n . \text{pre} = x$ [accessibilite]

(LS2) $(\exists n!N) \text{ non def sui}^n . \text{pre}$ [finitude]

Il est commode de regler immediatement les questions de consistance et d'indépendance d'axiomes en donnant des exemples pertinents.

Constructions diverses et axiomes satisfaits.



5.2.3. Principales propriétés.

Les deux axiomes ont la forme de quantifications existentielles. Il est intéressant de démontrer l'unicité. Dans ce genre de démonstration, on supposera connue l'arithmétique du premier ordre, dont tous les théorèmes sont aussi des théorèmes pour chaque structure chaînée.

La première démonstration d'unicité part de la propriété fondamentale du symbole "def":

- $\text{def sui } x \Leftrightarrow \text{def } x$ (vrai quel que soit le symbole sui)

Employe dans ce sens d'implication on en tire:

- $\text{def sui}^{n+1} x \Leftrightarrow \text{def sui}^n x$

- $\text{def sui}^n \text{ pre} \Leftrightarrow (\forall m \in \mathbb{N}) m(n \Leftrightarrow \text{non def sui}^m \text{ pre})$

Employe sous la forme contraposée on aura:

- $\text{non def sui}^n \text{ pre} \Leftrightarrow [(\exists m \in \mathbb{N}) m(n \Leftrightarrow \text{non def sui}^m \text{ pre})]$

On conclut alors par l'axiome de finitude:

- $(\exists ! n \in \mathbb{N}) \text{ def sui}^{n-1} \text{ pre et non def sui}^n \text{ pre}$

Cet unique entier n sera tout naturellement appelé longueur de la liste.

La deuxième démonstration d'unicité utilise aussi des propriétés arithmétiques:

- $\text{sui}^n x = \text{sui}^{n+k} x \Leftrightarrow \text{sui}^n x = \text{sui}^{n+mk} x$

$\text{sui}^n x = \text{sui}^{n+k} x \text{ et } k \neq 0 \Leftrightarrow (\forall l)(\exists m \neq 1) \text{ sui}^n x = \text{sui}^m x$

Le résultat obtenu étant incompatible avec l'axiome de finitude dans le cas $x = \text{pre}$ on déduit:

- $(\forall y) (!n \in \mathbb{N}) y = \text{sui}^n \text{ pre}$

Ceci définit une fonction d'ordre de l'ensemble des éléments de la liste dans l'intervalle initial $[0..longueur-1]$ des entiers, qui est une bijection.

Dans cette bijection les fonctions "sui" et $n \rightarrow n+1$ son associees ainsi que les constantes "pre" et zero. Il est interessant de se preoccuper des notions associees a la constante longueur-1 et a la fonction $n \rightarrow n-1$. La premiere est une constante partielle "dernier" et la seconde une fonction partielle "precedant", reciproque en un certain sens de "sui". Et ceci nous introduit la notion de liste "double".

5.2.4. Liste lineaire double.

On appellera liste lineaire double une structure chainee ayant pour squelette:

- un type unique
- deux moules de variables "pre" et "der"
- deux moules propres "sui" et "arr"
- un ensemble d'information quelconque

et pour axiomes:

- (LD1) $(\forall x) (\exists n!N) \text{ sui}^n \text{ pre} = x$
- (LD2) non def sui der
- (LD3) non def arr pre
- (LD4) $x \# \text{ der} \Leftrightarrow \text{arr sui } x = x$

On construit canoniquement comme on l'a vu une liste double a partir de toute liste simple, ce qui suffit a montrer la consistance. Pour l'indépendance on opere comme precedemment (voir figure).

La structure definie par pre et sui seuls est une liste lineaire simple, puisque (LD1) et (LD2) coincident tandis que (LS2) est consequence immediate de (LD2) et (LD1). La premiere propriete fondamentale est que der et arr constituent aussi une liste lineaire simple. Des qu'on aura montre (LS1) on aura aussi (LS2) consequence triviale de (LD3) et (LS1).

Demonstrons donc (LS1) pour der et arr. On commence par montrer l'écriture symetrique de (LD4).

- (i) $y \# \text{ pre} \Leftrightarrow (\exists x) y = \text{sui } x$ [par LD1]
- (ii) non $(\exists x) \text{ pre} = \text{sui } x$ [finitude]
- (iii) $y = \text{pre} \Leftrightarrow [(\forall x) y \# \text{sui } x]$ [par i et ii]
- (iv) $x \# \text{ der} \Leftrightarrow \text{sui arr sui } x = \text{sui } x$ [par LD4]
- (v) $x \# \text{ pre} \Leftrightarrow (\exists!y) x = \text{sui } y$ [LD1 et finitude]
- (vi) $\text{def sui } y \Leftrightarrow y \# \text{ der}$ [unicite deja montree]
- (vii) $x \# \text{ pre} \Leftrightarrow \text{sui arr } x = x$ [v, vi, iv]
- (viii) $\text{sui}^n \text{ pre} = \text{der}$ pour un certain n
- (ix) $\text{arr}^n \text{ sui}^n \text{ pre} = \text{arr}^n \text{ der}$
- (x) $\text{pre} = \text{arr}^n \text{ der}$

et en fixant x :

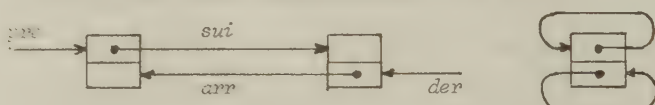
-(xi) $x = \text{sui}^m \text{pre}$ pour un certain m

-(xii) $x = \text{sui}^m \text{arr}^n \text{der}$

-(xiii) $x = \text{arr}^{n-m} \text{der}$

Ces propriétés permettent de ramener dans la quasi-totalité des cas les problèmes sur les listes doubles à deux problèmes sur listes simples.

Indépendance des axiomes de liste linéaire double.



non LD1



non LD2



non LD3



non LD4

5.2.5. Construction de listes linéaires chaînées.

Nous avons noté plus haut le lien essentiel entre la structure de liste au sens le plus large et la notion de balayage. Nous ne reviendrons donc pas sur l'utilisation de ce cas particulier des listes linéaires qui utilisent les structures chaînées comme formalisme. Inversement la question des opérateurs primitifs de construction nécessite un approfondissement.

5.2.5.1. "ajouter X derriere Y defini"

Dans cet operateur les arguments X et Y sont des references de reperes toutes deux definies. Soit alors la sequence d'operateurs primitifs (de structures chainees):

```
: modifier ( (X,sui) , (Y,sui) )  
: modifier ( (Y,sui) , X )
```

Pour demontrer une telle sequence, on introduit (cf 3.1.5) les operateurs elementaires sous la forme fonctionnelle $\langle R,C,I \rangle = f(\langle S,D,J \rangle)$ dans laquelle les triplets sont des constructions (cf. definition en 3.1.2). Ici nous aurons trois constructions successives $\langle S,D,J \rangle$ transformees en $\langle R,C,I \rangle$ par le premier operateur puis en $\langle Q,B,H \rangle$ par le second. La demonstration est sans difficulte:

```
: Q = R = S  
: H = I = J  
: B(Y,sui) = CX = DX  
: B(X,sui) = C(X,sui) = D(Y,sui)
```

Si la construction de depart est une liste lineaire simple, alors la construction finale en est une aussi sous les conditions suivantes:

- X et Y sont des references de reperes definies
- X n'appartient pas a la liste de depart

Cette demonstration met en avant un aspect permanent de l'utilisation des structures chainees. Un petit "secteur" est modifie, le reste demeurant invariant. Les demonstrations fondamentales seront souvent aussi aisees que la precedente concernant les modifications. Une difficulte notable, souvent laissee dans l'ombre, apparait dans les structures complexes. Il faut aussi demontrer que les operateurs primitifs ou construits ne modifient rien en dehors de leur secteur connu d'application. Reprenons notre exemple de 1.4 et 3.2. Il n'y suffit pas de montrer qu'un objet a ete range dans une liste, il est inevitable de montrer aussi que toutes les autres listes de meme nature sont invariantes par cet operateur d'ajout. Ceci impose donc une certaine contrainte supplementaire: le repere ajoute X ne doit appartenir a aucune de ces listes. Sans cette contrainte la demonstration est impossible.

Une contrainte homologue apparait des qu'on depasse le stade elementaire de la liste unique. Il est imperatif de realiser une demonstration semblable d'invariance. Elle n'est est possible qu'a condition que la valeur de la reference de repere $\langle X,sui \rangle$ soit a cet endroit du programme independante des axiomes de la structure. Notons qu'une telle contrainte est presque toujours satisfaite par un objet qui vient d'etre cree.

La nature de cette contrainte a des consequences importantes. La complexite de sa demonstration peut varier fortement d'un cas a l'autre. Le plus souvent il faudra se contenter de justifications informelles pour ne pas arreter la construction du programme. On n'est plus ici dans les conditions habituelles de preuve de programme, et on peut raisonnablement conjecturer qu'il sera impossible de realiser un systeme de preuve automatique acceptant les structures chainees dans toute leur puissance de realisation, ni d'ailleurs les autres formalisations possibles des pointeurs. En tout etat de cause il s'agit la d'une difficulte majeure.

5.2.5.2. "ajouter X en tete"

Ce second operateur primitif d'ajout est comparable au precedent. Le plus souvent il sera defini un operateur unique "ajouter X derriere Y" qui se reduit au premier ou au second selon que Y est defini ou non. Avec toutes les remarques et contraintes developpees precedemment, on montre sans peine la correction de la sequence:

```

:      modifier( (X,sui) , pre )
:      modifier( pre , X )

```

5.2.5.3. "retirer derriere Y defini"

Soit l'instruction:

```

:      modifier( (Y,sui) , (sui(Y),sui) )

```

On montre aisement qu'elle correspond effectivement a l'operateur cherche avec les contraintes d'emploi suivantes:

- Y est un repere defini de la liste
- sui(Y) est defini

Dans la majorite des cas il sera necessaire de chercher prealablement le precedent de l'element a retirer, mais les methodes en sont connues: moule convenable (liste double) ou algorithme de recherche avec conservation du precedent:

```

:      X := premier
:      PX non defini
:      TR := faux
:      tant que X defini et non TR faire
:      si propriete cherchee de X
:      alors TR := vrai
:      sinon PX := X ; X := suivant(X) finsi
:      fin tant que

```

Dans un grand nombre de cas concrets, l'isolement du repere X par rapport a la liste suffit a assurer l'indépendance de la valeur de $\langle X, \text{sui} \rangle$ par rapport aux axiomes. Notre réalisation évoquée au chapitre 7 contient un contre-exemple ou cela ne suffit pas et ou un algorithme complémentaire est nécessaire. Un tel algorithme a exactement le sens logique d'une destruction par rapport a la liste sans pour autant devenir une destruction physique du repere.

5.2.5.4. "retirer en tete"

Les commentaires sont les memes que pour l'operateur precedent. La contrainte est ici que la liste soit non vide. Comme pour l'ajout on disposera concretement d'un operateur unique "retirer derriere Y". L'operateur "retirer en tete" se demontre alors sous la forme classique:

```
! modifier ( pre , (pre,sui) )
```

5.2.6. Listes lineaires ordonnees.

Une liste lineaire simple ou double est dite ordonnee lorsqu'elle verifie deux axiomes supplementaires:

- (LO1) le symbole relationnel binaire $\ll$ definit un ordre total sur l'ensemble des informations
- (LO2) $\text{info} (X) \ll \text{info} (\text{sui} (X))$

Ce second axiome n'entraîne pas formellement l'axiome (LS2) ou (LO2) de finitude. Cependant pour tout modele fini verifiant (LO2), on aura aussi (LS2). Les axiomes introduits sont independants des precedents, mais le systeme obtenu n'est plus constitue en aucun sens satisfaisant d'axiomes independants.

On definit une notion moins forte de liste ordonnee suivant un element de l'information dans les conditions suivantes:

- l'ensemble d'informations est un produit $Z=H \times K$
- l'ensemble H est totalement ordonne
- $\text{info} (x) \ll \text{info} (\text{sui}(x))$

Cette situation couramment rencontree est un cas particulier de cas ou on imposerait seulement un preordre total sur les informations.

Ces deux structures derivees se traitent aisement a partir des operateurs de liste simple, a des details pres. On remarque tout d'abord que dans une liste ordonnee, l'ajout ne peut se faire qu'en un endroit univoquement defini. Il n'y a donc qu'un unique operateur "ajouter X en ordre". Cette unicite n'existe plus pour les listes preordonnees. Dans l'un et l'autre cas, certaines sous-listes peuvent etre definies par l'utilisation de l'ordre, notamment la sous-liste des elements dont l'information appartient a un certain intervalle de l'ordre ou du preordre. Il est commode alors de realiser les deux fonctions "premier" et "defini" associees pour rester dans le cadre habituel d'utilisation des listes.

5.3. LISTES BINAIRES (et K-aires)

Il apparait rapidement que la notion de liste binaire (sans ordre sur les informations) s'étend au cas de plusieurs successeurs pour chaque noeud. Nous definirons donc directement des listes K-aires. Dans cette definition K est un entier informel. En particulier tout axiome ou interviendra une "variable" de l'intervalle $[1..K]$ sera en realite un schema d'axiome (ou conjonction d'axiome puisqu'on est dans un cas fini). On pourra aussi trouver des disjonctions finies pour les quelles on emploiera la notation $\mathbf{W}$ [KRE67] et de meme $\mathbf{M}$ pour les schemas.

La notion de liste binaire est la structure chainee associee a la notion habituelle d'arbre binaire, c'est a dire une structure ressemblant a une arborescence binaire mais ou l'on est capable de distinguer les successeurs ("gauche" et "droit"). Par exemple les deux dessins suivants representent des objets identiques en tant que graphes, mais pas en tant qu'arbres binaires.

Lors de la definition habituelle intervient une notion d'accessibilite d'un sommet a partir d'un autre. Cette notion est par rapport aux structures chainees definissable au deuxieme ordre seulement. Comme nous l'avons indique precedemment (2.3.3), cela reflete la necessite d'un raffinement supplementaire. La methode habituelle utilise une pile, mais elle reflete mal la realite profonde de la structure. Nous pensons que la notion fondamentale est celle de "remontee", meme si elle peut souvent etre realisee avec seulement une pile.

5.3.1. Definition des listes K-aires.

On appelle liste K-aire la structure chainee qui a comme squelette:

- un type unique
- un ensemble quelconque d'informations
- un moule de variable "rac"
- K+1 moules propres "rem", "suc₁", "suc₂", ..., "suc_K"

et comme axiomes:

- (LM1) $(\exists n) (\forall x) \text{ non def rem}^n x$ [finitude]
- (LM2) $(\forall x) (\exists n) \text{ rem}^n x = \text{rac}$ [accessibilite]
- (LM3) $\mathbf{M} [(\forall z) (\forall m) (\forall n) (\text{rem } x = \text{suc}_a z \text{ et rem } y = \text{suc}_b z \Leftrightarrow x \# y)]$
 $a \# b$ [pas de cycles]
- (LM4) $x = \text{rac} \text{ ou } \mathbf{W}_a (\text{suc}_{\text{rem}} x = x)$ [reciproque]

La signification de ces axiomes peut être rendue plus intuitive en introduisant l'abréviation $ACC(x,y)$ avec la signification $(\exists n!N) (rem^n x=y)$. Compte tenu de la réciprocity (LM4) cette abréviation signifie informellement "x est accessible à partir de y en utilisant les fonctions suc_a ". Les axiomes signifient alors:

- (LM1) la longueur des chemins est bornée
- (LM2) tout est accessible à partir de la racine
- (LM3) on accède à tout repère par un chemin unique
- (LM4) "rem" est la fonction réciproque des " suc_a ".

Tout modèle est donc fini. Si on définit sur les repères un graphe en prenant tous les $(x, suc_a(x))$ pour arcs, on obtient une arborescence dont "rem" est la fonction prédécesseur. Mais cette affirmation est naturellement du deuxième ordre dans sa formulation.

Nous schématisons plus loin l'étude de consistance et d'indépendance d'axiomes.

5.3.2. Liste binaire ordonnée.

Dans le cas où l'ensemble des informations est muni d'un ordre total " $\leq$ " on définit une nouvelle structure, dérivée de la liste binaire, en remplaçant (LM3) par:

- (LB03I) $rem\ x = suc_1\ y \Leftrightarrow info(x) \leq info(y)$
- (LB03S) $rem\ x = suc_2\ y \Leftrightarrow info(y) \leq info(x)$

Il est clair que la conjonction de ces deux axiomes entraîne l'ancien (LM3).

5.3.3. Cas particulier où $K=1$.

Lorsqu'on envisage la liste 1-aire au sens défini en 5.3.1, on obtient la liste linéaire double définie précédemment, à la variable "dernier" près.

Démontrons rapidement:

- $x \# rac \Leftrightarrow suc\ rem\ x = x$ [par LM4]
- $(\exists n) rem^n x = rac$ [par LM2]
- $(\exists n) suc^n rem^n x = suc^n rac$
- $(\exists n) x = suc^n rac$ [le LD1 cherche]
- non def rem rac [LM1 et LM2 $\Leftrightarrow$ LD3]
- $[(\exists y) x = rem\ y] \Leftrightarrow rem\ suc\ x = x$ [par LM4]
- $(\forall n)(!x) rem^n x = rac$ [c'est $suc^n \cdot rac$]
- $(\exists n) non\ def\ suc^n\ rac$ [le LS2 cherche]

Cette definition de liste lineaire comme cas particulier de liste K-aire est purement theorique. Les methodes d'utilisation des listes lineaires ne se transmettent pas aux autres listes, et inversement les algorithmes sur listes K-aires perdent tout interet appliques aux listes lineaires. A partir de ce moment nous excluerons donc de nos preoccupations le cas $K=1$.

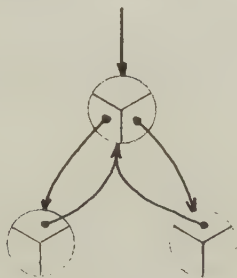
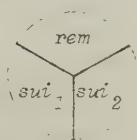
5.3.4. Exemple concret de K-aire non binaire.

Il parait utile d'indiquer dans ses grandes lignes ce que peut etre une liste K-aire avec $K \geq 2$, pour indiquer que ce n'est pas seulement une speculation theorique. Prenons un alphabet A de K elements et un langage fini L sur l'alphabet A. On suppose que L a la propriete que si un mot est dans L, alors tout segment initial y est aussi. Notons rac le mot vide, " suc_a " la fonction consistant a suffixer un mot par la lettre "a", et enfin " rem " la fonction qui enleve la derniere lettre. Il n'y a aucune difficulte a concevoir ni a demontrer que l'on obtient ainsi une liste K-aire, ceci quel que soit K (y compris 1 et 2).

Une telle structure est un moyen parmi d'autres de represente une table. Elle a un defaut essentiel dans l'utilisation de la place. En effet le nombre de references de reperes definies ne depend pas de K. Donc a l'inverse le nombre des references non definies augmente avec K. L'utilisation concrete de listes K-aires semble donc de plus en plus improbable lorsque K augmente.

En revanche il est toujours possible de considerer comme liste K-aire la meme structure dans laquelle les " $\text{suc}_a(x)$ " pour x donne sont organisees en listes lineaires simples ordonnees. On passe de la premiere a la seconde forme en redefinissant convenablement les fonctions " suc_a " et " $\text{modifier}(*, \text{suc}, *)$ ". Dans ce cas l'axiomatique seconde forme peut exprimer directement l'idee que ces nouvelles fonctions organisent une liste K-aire.

Consistence et indépendance pour la liste binaire.



Consistence



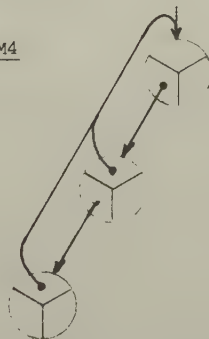
non LM1



non LM2



non LM3



non LM4

Toute autre situation est interdite ("rem" est fonctionnelle).

5.3.5. Proprietes et utilisation des listes K-aires.

Etant donne une liste K-aire (rac, rem, suc_a) et un element x quelconque d'une telle liste, le triplet (x, rem', suc_a) organise en liste K-aire les elements y caracterises par $ACC(x, y)$, en prenant pour "rem'" la fonction non definie en x et coincidant avec "rem" ailleurs. On peut aussi la considerer comme la restriction de "rem" au nouvel ensemble de valeurs, l'absence de cycles entrainant sa non definition en x .

Cette propriete permet de definir recursivement les listes K-aires. Une liste K-aire est alors un K-uplet dont les elements sont soit une liste plus petite, soit vides. Inversement nous ne pouvons nous contenter de definir une liste K-aire comme un element du plus petit ensemble H stable par la fonction de construction de K-uplets sur $H \cup \{\text{vide}\}$. Une telle definition n'elimine pas les cycles et ne permet pas de definir "rem" comme application univoque. Par contre il est possible de definir une operation de construction de K-uplets dont les elements sont soit vide, soit une tete de K-uplet existant. Tout objet complexe construit par une suite d'appels d'une telle operation elementaire est alors une liste K-aire. Il faut noter que c'est la sequentialite des operations qui permet d'obtenir le resultat. Cette definition est purement informatique (dynamique) tandis que la structure chainee est statique.

L'utilisation d'algorithmes recursifs sur les listes K-aires est possible. Il s'agit en general d'algorithmes de la forme:

$$\begin{aligned} f(\text{vide}) &= f_0 \\ f(\langle a_1, \dots, a_K \rangle) &= F(f(a_1), \dots, f(a_K)) \end{aligned}$$

L'appel recursif porte donc successivement sur tous les successeurs ce qui rend delicate l'elimination de la recursivite, tout au moins en principe. Pour passer d'un algorithme recursif a un algorithme non recursif, on remarque que d'apres l'axiome (LM4) tout sommet est soit la racine soit le i -ieme successeur d'un autre sommet et d'apres (LM3) il n'est que dans une de ces $K+1$ situations. Dans l'elimination de la recursivite une telle information a la meme nature qu'une adresse de retour. L'algorithme obtenu par un tel procede ressemble tout naturellement a un balayage pour l'ordre postfixe sur les elements de la liste.

5.3.6. Exemple de demonstration d'algorithme sur les listes.

Nous allons montrer que notre axiomatique sur les listes binaires ordonnees suffit pour demontrer l'algorithme classique de localisation (recherche). Nous le rappelons sous la forme suivante, ou nous utilisons l'affectation au lieu de "modifier". Cette utilisation de variables non formelles dans un algorithme sur les structures chainees est correcte. Elle exprime l'idee que nous disposons d'une structure de base dont nous connaissons l'axiomatique et que nous ne modifions pas. On peut alors realiser des demonstrations a partir de cette axiomatique et de celle des structures de controle, dans les memes conditions qu'avec un programme d'arithmetique, et en particulier utiliser toutes les methodes connues dans ce domaine.

```

:   PX := nil ;
:   x := rac ;
:   T := faux ;
:   S := SRAC ;      etype scalaire SRAC,SINF,SSUPe
:   tant que (X defini) et non T faire
:       si info(X)=cherche alors T := vrai sinon
:           debut
:               PX := X ;
:               si info(X) < cherche alors debut X:=suc1(X); s:=SINF fin
:                   sinon debut X:=suc2(X); S:=SSUP fin
:           fin

```

L'algorithme donne definit deux suites de reperes $PX_0, PX_1, \dots$ et $X_0, X_1, \dots$ de meme longueur, une suite $S_0, S_1, \dots$ de valeurs scalaires, une suite $T_0, T_1, \dots$ de booleens. On a les proprietes suivantes, liees aux structures de controle:

- $X_0 = \text{rac}$ et $PX_0 = \text{nil}$ et $S_0 = \text{SRAC}$
- $PX_{i+1} = X_i$ si T_{i+1} est faux
- X_{i+1} est successeur de X_i , du cote defini par S_{i+1}
 si T_{i+1} est faux et X_i est defini

D'apres les axiomes de la structure on a donc tant que T_{i+1} est faux et que X_i est defini:

- $\text{rem}(X_i) = X_0 = \text{rac}$

En particulier un tel i est borne donc l'algorithme s'arrete.

Lorsque l'algorithme s'arrete, ou bien X est defini ou bien il ne l'est pas. Dans le premier cas T est vrai, donc X porte l'information cherchee, X est successeur de PX du cote defini par S (avec le cas particulier (nil,SRAC)). Sinon T est faux, on peut alors affirmer (par LB03I et LB03S) que l'element cherche ne peut se trouver que dans les sous-listes ou il a ete cherche (sous-listes du cote S_{i+1} par rapport a X_i). Donc a la fois il n'existe pas et on cree une liste ordonnee correcte en ajoutant la sequence:

```

:   creer(Y)
:   valeur(Y,cherche)
:   selon S parmi SRAC : modifier(rac,Y)
:                       SINF : modifier((PX,suc1),Y)
:                       SSUP : modifier((PX,suc2),Y)
:
:   fin de selection

```

Si le nouvel element n'est pas cree a ce moment, mais qu'on insere seulement un element existant, il faut ici encore introduire la contrainte d'indépendance des valeurs des references de reperes modifiees (cf 5.2.5) et eventuellement assurer leur non-definition.

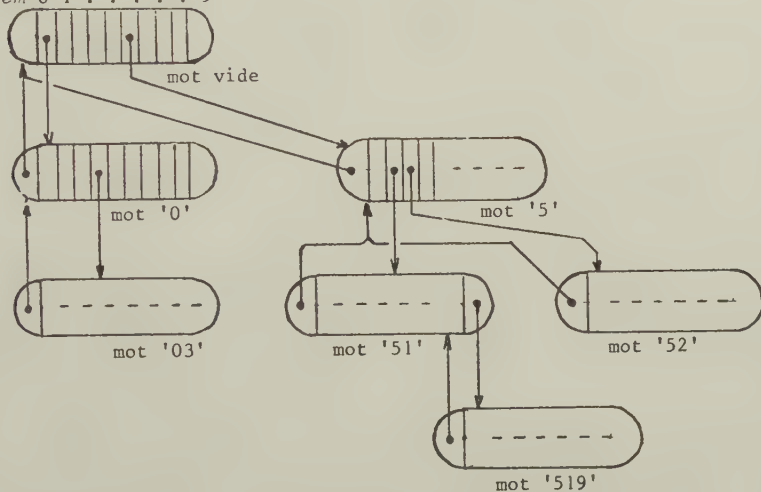
L'axiomatique donnee pour les listes binaires ordonnees suffit donc pour prouver l'algorithme classique de recherche/ajout. On peut ajouter que le retrait de l'un quelconque des quatre premiers axiomes rend caduc l'algorithme:

- sans (LM1) on ne peut affirmer l'arret de l'algorithme
- sans (LM2) l'element cherche peut exister sans etre trouve par l'algorithme
- sans (LB03I) ou (LB03S) on ne peut plus discriminer entre les sous-arbres celui qui seul peut contenir l'element cherche
- sans (LM4) on ne peut plus definir l'accessibilite (tout au moins en logique du premier ordre).

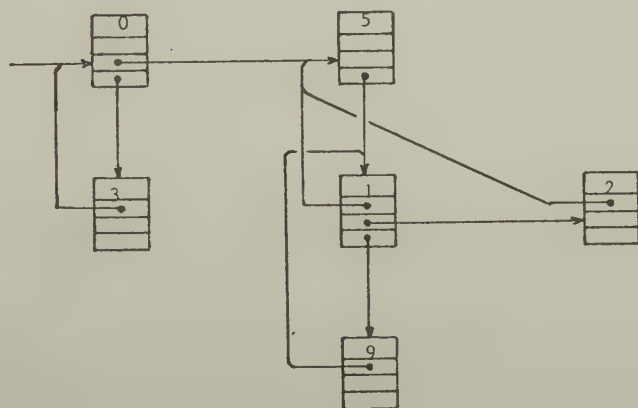
Nous pouvons conclure a une bonne adequation de la structure chaine de liste lineaire ordonnee vis-a-vis des buts recherches.

Exemples de listes 10-aires.

rem 0 1 9



78 "places"
25 "places"



ADAPTATION DU LANGAGE PASCAL

AUX STRUCTURES CHAINEES

6.1. GENERALITES

L'etude des langages existants a montre leur inadaptation parfois difficilement surmontable a la representation des structures chainees dans toute leur generalite. Il est parfois necessaire de recourir a des langages de niveau inferieur. Cette contrainte n'a aucun caractere obligatoire, et nous allons donner un certain nombre de propositions justifiant cette opinion.

Dans ce chapitre nous allons ainsi definir diverses structures de langage, avec a chaque fois une proposition de forme syntaxique, son contenu semantique, une voie de compilation et realisation raisonnablement simple. La forme syntaxique suppose toujours comme base minimale le langage Pascal, et les donnees de compilation se referent au compilateur pour la machine virtuelle P-Code [GAU74].

Ce serait une erreur de considerer les langages a types abstraits comme etrangers au present chapitre. A notre sens aucun langage ne pourra se cantonner au seul niveau de definition des types abstraits sous peine d'une rigidite redhibitoire. Il sera donc toujours necessaire de donner au programmeur les moyens de representations a la fois simples et efficaces. De bons operateurs de structures chainees sont un de ces moyens.

6.2. REDEFINITION DE LA COMPATIBILITE DES TYPES

6.2.1. Situation existante.

La situation actuelle est assez confuse. L'equipe de conception du langage Pascal indique elle-meme plusieurs choix possibles. En dehors des intervalles, toujours compatibles avec leur type sous-jacent, on rencontre deux cas extremes parmi les compilateurs disponibles:

- deux types sont compatibles uniquement s'ils ont le meme nom
- deux types sont compatibles avec une definition recursive tres liberale:

- = deux articles sont compatibles lorsque les deux suites de types des champs (dans l'ordre de leur declaration) sont composees d'elements deux a deux compatibles dans le meme ordre
- = les articles avec variantes sont incompatibles
- = deux tableaux sont compatibles lorsque les types de leurs elements le sont, avec trois choix differents pour les types d'indices.

Le premier choix est manifestement inadapté a la programmation structuree et doit etre fermement rejete. En ne permettant pas que deux types abstraits aient la meme representation, on introduit des renominations qui rendent le programme difficile a mettre au point, a modifier, a comprendre. Le second choix permet par sa souplesse un raffinement aise mais n'offre pas suffisamment de possibilites de controle au programmeur.

En realite il est raisonnable de demander que deux types soient compatibles pour certaines procedures et incompatibles pour d'autres. Par exemple deux types peuvent etre differents au niveau de leur utilisation mais realises de la meme facon et donc identiques a d'autres niveaux. Second exemple, il est souhaitable que les types d'une "union" puissent etre distingues dans leur utilisation mais reunifies pour les procedures d'allocation et desallocation.

Par ailleurs B. Liskov [LIS78] a montre l'interet de types dont l'ensemble des operateurs est restreint par rapport au type principal sous-jacent. Un tel choix est bien adapte aux structures chainees, le type restreint correspondant alors a une sous-structure particuliere.

6.2.2. Arborescences de types.

Nous proposons de completer les langages en adjoignant une nouvelle forme de declaration de types, que nous presentons ici par son diagramme syntaxique (ou de Conway):



Semantiquement la declaration "T(U)" est valide des que U est un type declare par ailleurs (precedemment ou non selon le choix general du langage). Elle realise la creation d'un lien ordonne entre T et U, l'ensemble de ces liens structurant l'ensemble des types en une foret d'arborescences. Les racines en sont ainsi les types ayant une declaration explicite (au sens habituel).

Deux structures se surajoutent ainsi: une relation d'egalite definie par les declarations "T=U", pour laquelle le type ayant la declaration explicite conserve cependant un caractere privilegie; et une seconde relation definie par les arborescences. La compatibilite entre deux types T1 et U1 se calcule comme suit. On cherche pour T1 et U1 les types privilegies associes T2 et U2. On calcule ensuite si l'un ou l'autre de T2 et U2 est antecedent du second, ce qui est l'unique cas de compatibilite. Il ne semble pas necessaire d'elargir cette condition dans la mesure ou les utilisations habituelles de la compatibilite la plus liberale se raffinent aisement dans la nouvelle forme. Par ailleurs la definition d'un tel elargissement est tres delicate pour etre compatible avec l'arborescence des types.

Cette structuration des types s'integre aisement a un compilateur Pascal existant: dans l'article associe a chaque type on introduit deux nouvelles variantes (pour les types declares par egalite et par arborescence); le pointeur associe donne alors selon le cas le type appele plus haut privilegie ou bien le predecesseur dans l'arborescence. L'algorithme de compatibilite est alors lineaire en fonction de la profondeur des elements compares dans l'arborescence.

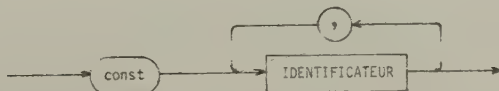
6.2.3. Restriction des operateurs disponibles.

Nous n'envisageons ici que la restriction des operateurs pour les articles. Seule une partie des possibilites de restriction nous semble fondamentale, celle qui permet de travailler sur les sous-structures. En d'autres termes nous proposons d'expliciter les champs autorises de l'article en distinguant les champs en lecture et les champs modifiables.

Ceci peut se decrire par une nouvelle forme de declaration de type definie comme suit:



liste const



liste var



Cette declaration est valide lorsque l'identificateur entre "=" et "[" (celui du type principal) est declare ailleurs dans le meme bloc ou un bloc englobant, et que les identificateurs des deux listes sont distincts et pris parmi les noms des champs du type principal.

Cette nouvelle forme de declaration n'introduit pas de type reellement nouveau. C'est une forme plus generale que la simple egalite. Les operateurs de creation, destruction et affectation ont une definition fonctionnelle inchangee. Par contre leurs conditions de validite sont restreintes a l'emploi des champs explicitement redeclares. Tout acces est admis pour un champ redeclare "var" mais seule la lecture est autorisee pour un champ "const".

Un tel controle est realisable statiquement. On introduit d'abord pour les champs un boolean const/var indiquant les acces autorises. Pour une declaration classique, la valeur par default est naturellement "var". Le nouveau type correspond a une nouvelle variante dans les tables du compilateur, laquelle reference le type sous-jacent et un nouveau dictionnaire des champs autorises. Ceci definit completement les tables modifiees.

L'interdiction d'emploi des champs non explicitement redeclares se realise en ne les rencontrant pas dans le dictionnaire des champs pour ce type. Les restrictions sur la modification ne portent que sur deux especes d'occurrences du nom de champ: en fin d'une variable soit a gauche dans une affectation, soit comme argument par reference d'une procedure.

Les modifications du compilateur sont donc ici nettement plus lourdes. Leur cout nous semble acceptable compte tenu des avantages apportees a un controle fin de la programmation.

6.2.4. Combinaison des deux propositions.

La combinaison de ces deux structures nouvelles du langage n'est possible qu'en imposant une hierarchie entre elles. Nous proposons simplement de considerer les types restreints comme des racines de nouvelles arborescences de types. Dans ces conditions les deux modifications du compilateur sont compatibles.

6.3. GESTION FINE DES VARIABLES

6.3.1. Situation actuelle.

On connaît actuellement trois systemes principaux de gestion des variables. Celui de Fortran connaît les variables au niveau de chaque sous-programme. Celui des langages a structures de blocs definit une portee egale au bloc, moins eventuellement le ou les blocs englobes ou est redeclare le meme identificateur. Le troisieme complete la structure de blocs par un mecanisme de "modules" ou "moniteurs" ou "classes" permettant la gestion sure et efficace des variables communes aux operateurs de representation d'un type abstrait. Notre exemple de la section 1.4 montre assez bien une certaine insuffisance de finesse des deux premiers choix. Les diverses variables ont souvent une portee logiquement plus restreinte que le bloc de leur declaration. C'est une source de clarte et de controle que de definir dans le programme la portee minimale pour chaque variable, et ceci n'est pas particulier a la programmation avec les pointeurs. Les propositions que nous faisons completent la gestion par blocs et peuvent donc s'ajouter aussi a un systeme mixte blocs/modules.

6.3.2. Differents statuts de variables.

L'etude des diverses utilisations de variables met en lumiere des statuts differents d'utilisation des variables:

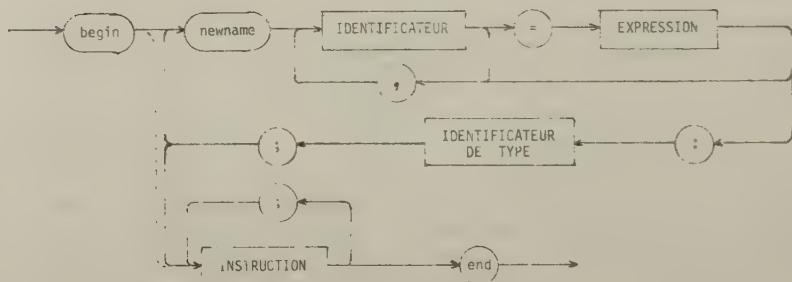
- variables possedant une unique affectation a l'interieur d'une structure de programme, et ne transportant pas de valeurs a l'exterieur de cette structure (iteration ou alternative)
- variables transmettant des valeurs d'un pas a un autre d'une iteration
- variables de commande d'iterations, qui sont necessairement des variables muettes (ce qui suppose une instruction explicite de balayage).

Ces divers statuts peuvent efficacement etre rendus explicites au niveau des langages.

Les variables de la premiere categorie ont une portee logique qui est celle de l'instruction commandee, celles de la troisieme correspondent a l'iteration dans son entier. Dans les deux cas la portee est une instruction. Pour la deuxieme categorie il est en theorie possible d'ajouter ou supprimer un moule de variable a tout moment. Cependant il n'est possible que d'ajouter un moule inconnu au moment considere et de supprimer un moule existant, et cette restriction doit etre controlable statiquement puisqu'elle exprime une propriete du squelette. Cela implique de lier l'ajout et la suppression aux structures de controle donc en fait ici encore a une instruction, eventuellement composee. Il est clair que cela ne permet que d'approcher la portee logique reelle des variables, que ce n'est qu'un compromis entre la theorie et la compilation.

6.3.3. Instructions declaratives.

Nous introduisons sous le nom d'instruction declarative une nouvelle forme d'instruction avec la syntaxe suivante:



En ajoutant le lien en pointille on insere l'instruction composee classique comme cas particulier, mais il importe de separer les deux cas, qui correspondent a des semantiques nettement differentes, de meme par exemple qu'en Algol-60 le bloc et l'instruction composee.

Cette syntaxe, qui rappelle le bloc au sens Algol-60, en differe pourtant sensiblement par son caractere statique. En l'absence de declaration de tableaux dynamiques, les instructions declaratives donnent seulement au compilateur des directives de portee des variables. Il peut en profiter pour affiner sa propre allocation, choix qu'il fera en general par simplicité et efficacite.

La definition semantique la plus simple procede par transformation. Considerons la forme la plus generale d'instruction declarative:

```

: begin
: newname ID1,1 { =EX1,1 },..., ID1,1 { =EX1,1 } : TY1 ;
:
: . . .
: newname IDm,1 { =EXm,1 },..., IDm,p { =EXm,p } : TYm ;
: IN1 ; ... ; INn end

```

Elle sera traitee exactement comme l'unique appel de la procedure suivante:

```

: var ID1,1 , ... , ID1,1 : TY1 ;
:
: . . .
: IDm,1 , ... , IDm,p : TYm ;
: begin
: ID1,1 := EX1,1 { des que EX1,1 existe } ;
:
: . . .
: IDm,p := EXm,p { des que EXm,p existe } ;
: IN1 ; ... ; INn end

```

ceci sous reserve que les instructions IN_i n'utilisent les variables preaffectees que comme constantes.

La compilation d'une instruction declarative utilise pour l'essentiel des structures existantes du compilateur primitif: gestion des listes de variables au niveau de la declaration et attribution du type differee en fin de liste; gestion d'une pile de dictionnaires. La seule nouveaute est l'adjonction pour les variables d'une information d'interdiction d'affectation, que l'on teste pour les affectations et les arguments par reference. Cette information est analogue a celle deja envisagee pour la restriction des operateurs lies aux champs des articles.

6.3.4. Balayage explicite des listes.

Soit la syntaxe



Cette écriture est valide quand le premier identificateur et la première expression sont d'un même type pointeur sur un article, que le second identificateur est un champ pointeur du même type dans ce même article et que l'expression est booléenne.

Si ces conditions sont définies l'écriture

```

:   for P := EX via CH while VB do IN
:   est équivalente à
:
:   begin
:   P := EX ;      { ou "newname P = EX : type" }
:   while (if P=nil then false else VB) do
:       begin IN ; P := P↑.CH end
:   end
  
```

ou P est interdit d'affectation dans IN.

Cette instruction se compile avec les structures déjà définies, puisqu'il s'agit en fait d'une macro-instruction. Si on a choisi de ne pas traiter les instructions déclaratives, elle se compile comme le "for" classique.

6.4. GESTION DES MOULES PROPRES

6.4.1. Principe general.

Nous avons introduit dans la definition des structures chainees des operateurs d'addition et de suppression de moules. Leur realisation avec toute leur puissance theorique serait assez complexe et n'est sans doute pas indispensable. Leur raison d'etre est la formalisation de l'idee courante de chainage temporaire, utilise pour la duree d'un algorithme. Ajouter un moule de variable ou un moule propre est en cela peu different. Cependant leur gestion est sensiblement plus complexe, ce qui nous conduit ici a une proposition quelque peu rigide. Malgre cette rigidite tous les cas classiques y trouveront une bien meilleure realisation qu'avec les langages habituels. Nous introduisons donc la possibilite de declarer des moules nouveaux pour la duree d'une procedure, ce qui revient fondamentalement a les creer a l'entree d'un algorithme et les detruire a la sortie. Il peut certes sembler souhaitable de transmettre un tel moule d'un algorithme a un autre, mais nous souhaitons fortement ne pas introduire trop de controles dynamiques, ce qui impose la propriete faite.

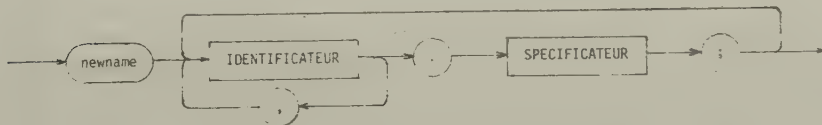
6.4.2. Mise en oeuvre de l'addition de moules.

Nous nommerons "specificateur" la regle syntaxique suivante:



Dans cette ecriture le premier identificateur designe un type (autre qu'une file) et le second un type article, tous deux definis par ailleurs.

Nous introduisons entre la partie "types" et la partie "variables" de la definition du bloc une nouvelle partie "champs" avec la syntaxe suivante:



Cette écriture déclare un ou plusieurs champs du type indiqué (par le spécificateur) dans l'article indiqué (idem). Ces champs nouveaux existeront pour la durée de la procédure ou ils sont déclarés. Ils doivent avoir la valeur "nil" des l'entrée de la procédure, et c'est naturellement la principale contrainte de programmation. Si l'initialisation a "nil" n'est pas effectuée à la création, on pourra aussi laisser au programmeur celle des champs nouveaux, sinon les outils du chapitre 4 donnent des moyens simples de réalisation. Les champs ainsi créés sont détruits à la fin de la procédure, ce qui autorise le compilateur à réutiliser la "place" correspondante pour d'autres champs temporaires.

La compilation d'une telle structure pose des problèmes inhabituels. En particulier elle rompt avec les particularités qui permettent la destruction simplifiée des tables d'un bloc (cf. 3.3.2). D'une part certains pointeurs de ces tables peuvent maintenant référencer un élément d'un bloc plus profond, d'autre part les repères associés à un bloc peuvent ne plus correspondre à un intervalle dans l'ordre total sur les repères. Il faut donc étudier spécifiquement la question des tables du compilateur, la génération de code ne subissant pas de changement.

6.4.3. Compilation des moules temporaires.

La solution proposée ici est sans doute assez lourde, mais elle semble bien adaptée à la compilation des notions associées aux moules temporaires, sans pour autant renoncer aux principes de simplicité rappelés au paragraphe précédent.

Le compilateur créera des copies dans ses tables pour chaque article subissant une addition de moule. Ce nouvel objet contiendra un dictionnaire complet des champs pour son niveau et, en fonction d'une variante booléenne le caractérisant comme copie, référencera son article original. Il suffit alors de gérer les adresses relatives des champs, par les techniques classiques.

Les articles avec variantes sont exclus de la possibilité d'addition de champs temporaires. Pour les autres la longueur du type n'est connue qu'en fin de compilation et doit donc être rangée en début d'exécution dans une variable prévue à cet effet.

L'obligation de réinitialiser les champs (à "nil" pour les pointeurs, facultative pour les autres types) est très contraignante. Elle impose au compilateur et à la bibliothèque d'exécution une organisation des repères alloués permettant leur énumération à tout moment. Dans un langage qui n'initialise pas automatiquement les champs il est cohérent (bien que peu correct) d'admettre que le programmeur s'organise comme il veut. Il est préférable d'offrir au programmeur

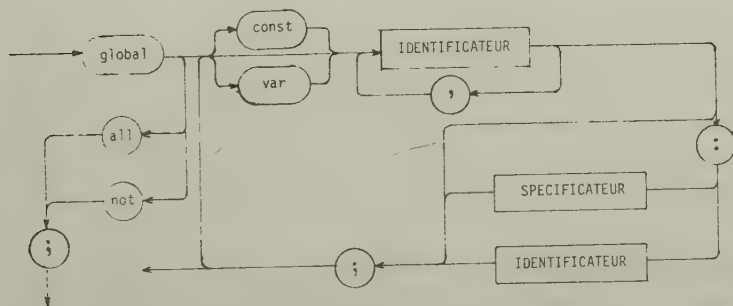
l'accès aux listes d'objets de chaque type, sans bien entendu qu'il puisse les modifier. Ceci peut se réaliser par des fonctions predeclarees FIRSTOC(type) et SUCC() existante pour les scalaires.

6.4.4. Declaration des globaux.

Un algorithme de maniere generale parcourt une certaine structure preexistante sans la modifier et cree ou modifie une autre sous-structure. Les algorithmes "non-numeriques" sont de cette forme a des variantes de detail pres. Il y est donc important pour une programmation sure d'avoir les moyens de dire explicitement sur quelle sous-construction, ou plus fondamentalement sur quel sous-squelette, on va travailler sans modification et quel(le) autre sera autorise(e) a etre modifie(e). Ces moyens doivent naturellement etre accompagnes des controles correspondants, realises autant que possible statiquement.

Dans les langages a structure de blocs, le moyen le plus simple est la declaration obligatoire des variables non-locales pour chaque procedure. On ne peut se contenter d'interdire certains identificateurs, il faut obliger la redeclaration explicite. Lors de la modification d'un langage existant, tout ancien programme doit demeurer avec la meme semantique, ce qui impose l'autorisation d'accès aux non-locaux en cas de non-redeclaration. Pour un langage nouveau, il est preferable de prendre des le depart l'option inverse.

Concretement la partie "variables" des declarations sera precedee d'une partie "globaux" comme suit:



Cette redeclaration est applicable aux variables et aux champs d'articles. On peut annuler une option par défaut et la remplacer par l'option inverse par la forme "global all" ou "global not". Pour les globaux redefinies explicitement on doit preciser si seul l'accès en lecture est admis ("const" ou absence d'indication) ou si la modification est autorisee ("var").

Cette forme n'est pas identique a ce qu'on peut obtenir avec les types restreints. Avec ceux-ci en effet un ensemble limite de champs est autorise pour les variables declarees avec ce type, mais non pour d'autres variables. Avec les non-locaux explicitement redeclares toutes les variables, y compris non-locales redeclares, ont leur ensemble de champs accessibles qui est restreint. Les deux formes se complementent ainsi sans rendre l'autre inutile.

Cette structure se compile avec les memes methodes qu'en 6.4.3. Chaque nouveau bloc est compile avec un dictionnaire completement recalculé ou seuls apparaissent les arguments, les locaux, et les non-locaux redeclares. Les methodes habituelles d'adressage des variables non-locales ou globales demeurent, le code objet étant dans tous les cas inchangé. Il y a seulement renforcement des controles.

6.4.5. Moules comme arguments.

Nous avons souligne au chapitre 4 la necessite de transmettre des moules comme arguments de procedure, par exemple pour ecrire des procedures de gestion de listes ou de toute autre sous-structure. Pour cela on autorisera dans les listes d'arguments de procedure ou fonction le remplacement de l'identificateur de type par un specificateur (tel que defini en 6.4.2). Un tel champ pourra ou non etre autorise a la modification ("const"/"var"). La notion associee a un tel argument dans le code-objet sera tout naturellement l'adresse relative du champ, notion machine sans difficulte.

6.4.6. Remarque sur la declaration des files.

L'obligation d'explicitement les non-locaux s'applique en particulier aux files comme variables et au programme principal comme bloc, ce qui fournit un cadre formellement acceptable pour une declaration "propre" des files avec leur interface systeme.

6.4.7. Un exemple d'utilisation.

Nous donnons ci-dessous la declaration de deux procedures de gestion de liste dans le langage modifie selon nos propositions precedentes.

```
: procedure AJOUT(X,{ derriere }Y:PNT; var PRE:PNT;
:       var SUI:field PNT of ART) ;
:       procedure AJOUTSIMPLE(var REF:PNT) ;
:       global X:PNT; var SUI:field PNT of ART;
:       begin X↑.SUI := REF; REF := X end ;
: begin
: if Y=nil then AJOUTSIMPLE(PRE)
:       else AJOUTSIMPLE(Y↑.SUI)
: end ;
: procedure RETRAIT({ derriere }Y:PNT; var PRE:PNT;
:       var SUI:field PNT of ART) ;
:       procedure RETRAITSIMPLE(var REF:PNT) ;
:       global SUI:field PNT of ART;
:       begin REF := REF↑.SUI end ;
: begin
: if Y=nil then RETRAITSIMPLE(PRE)
:       else RETRAITSIMPLE(Y↑.SUI)
: end ;
```

La situation serait encore meilleure si on pouvait donner au compilateur l'ordre de traiter AJOUTSIMPLE et RETRAITSIMPLE comme macro-instruction, mais ceci sort completement du cadre de notre travail.

6.5. ADDITIONS DIVERSES

6.5.1. Verification des dereperages.

Les operateurs elementaires sur les constructions sont definis souvent a partir de la notion de reference de repere, donc de couple (repere,moule). Pour satisfaire cette definition le repere (qui sera necessairement porte par une variable) doit etre defini. Nous considerons comme important de realiser dynamiquement ce controle du dereperage, ce qui n'est pas tres different du test de debordement d'indice pour les tableaux.

Certaines de nos additions precedentes permettent une certaine simplification en donnant au compilateur des directives utiles. Dans le cas du balayage "for P:= ... via ..." il est inutile de tester le dereperage de P des qu'on est certain de sa non-affectation, ce que nous proposons de controler statiquement. Dans le cas de variables d'affectation unique, compte tenu ici encore des controles statiques, on peut se contenter d'une verification unique a l'affectation. Une telle variable peut alors etre traitee dans les demonstrations comme une constante de repere, ce qui simplifie notablement.

6.5.2. Ajout et retrait de types.

Nous avons prevu dans la definition formelle que des operateurs permettent d'ajouter ou retirer des types. Comme pour les moules propres cette addition peut se faire pour la duree d'une procedure.

Nous ne la mentionnons ici que pour memoire, le probleme concret etant celui de l'addition de champs de ce type temporaire, lequel a ete resolu plus haut.

TRAITEMENT COMPLET D'UN EXEMPLE

7.1. ENONCE ET FORMALISATION

7.1.1. Expose informel de la situation.

Bien souvent les textes sur les methodes de programmation ne traitent que des exemples d'ecole, a la fois quasi-triviaux et ne montrant pas leur applicabilite a des situations concretement rencontrees ou a des problemes plus complexes. Nous avons montre dans la section 1.4 que des problemes tres simples engendraient deja des difficultes notables.

Il nous apparait donc necessaire de montrer que les structures chainees constituent un type abstrait et soutiennent une methodologie effectivement utilisables pour des cas non triviaux. Nous avons pris pour cela un probleme reel et non cree pour la circonstance, de taille a priori significative. Ce probleme est traite de bout en bout dans ce chapitre, quoique en resume. On trouvera en annexe des extraits du programme et des resultats. Un rapport technique et pedagogique detaille est prevu par ailleurs.

Le programmeur qui utilise Fortran rencontre souvent la difficulte suivante: il connait parfaitement pour chaque sous-programme quels autres il appelle, mais il a bien du mal a savoir par qui il est appele. En poussant un peu la reflexion, on s'apercoit que bien des informations utiles sont donnees par le calcul du "graphe des appels" entre sous-programmes:

- recherche de circuits
- stratification (au sens des graphes sans circuit)
- preparation d'une structure de recouvrements ("overlay")

Nous nous fixons ici pour but l'etablissement d'un programme de traitement des liens entre sous-programmes et fournissant:

- des references croisees
- la stratification dans les deux sens de parcours
- des references croisees "generalisees" (a partir de la fermeture transitive)
- la liste complete des circularites d'appel eventuelles.

7.1.2. Formalisation du probleme.

On trouve au depart une notion de nom. Un nom est a priori un identificateur au sens de Fortran. Toutefois dans le cas de sous-programmes en assembleur, leur syntaxe peut etre eventuellement moins rigide. On appellera donc "nom" tout mot de 1 a 8 caracteres pris parmi: les lettres majuscules, les chiffres, et des symboles divers pouvant varier d'une machine a une autre.

Pour ce qui nous interesse, un jeu de donnees est compose d'un ensemble non vide de sous-programmes. Chacun d'eux est muni d'un nom, d'un ensemble non vide de noms (points d'entree) et d'un ensemble eventuellement vide de noms (references externes). Deux sous-programmes (resp. deux entrees) ne peuvent avoir le meme nom.

On peut donc formaliser le probleme en termes ensemblistes comme suit:

- un ensemble de noms $NM = CAR$ donc fini
- trois ensembles disjoints P, E, X finis
- des fonctions de nomination NP (resp. NE, NX) de P (resp. E, X) dans NM
- une fonction EP de E dans P signifiant " $EP(e)$ est le programme de definition de e " et sa reciproque PE de P dans les parties de E .
- une fonction XP de X dans P signifiant " $XP(x)$ est le programme ou apparait la reference x " et sa reciproque PX de P dans les parties de X
- une fonction XE de X dans E signifiant " $XE(x)$ est l'entree de meme nom que x " et sa reciproque EX de E dans les parties de X
- il existe des entrees (elements de E) sans programme associe, soit temporairement (en cours de calcul) soit definitivement (acces a la bibliotheque).

On definit ensuite, avec P comme ensemble de sommets, un graphe dont les arcs verifient les conditions equivalentes:

- p appartient a $XP(EX(PE(q)))$
- q appartient a $EP(XE(PX(p)))$

A partir de ce graphe les algorithmes utiles sont connus: stratification et fermeture transitive sont dans tous les manuels elementaires, mais l'enumeration des circuits [TIE70] est moins connue.

Le choix d'enumerer les circuits peut sembler etonnant. Bien que l'algorithme soit efficace en fonction du nombre de circuits elementaires, ce nombre croit exponentiellement en fonction des sommets. Dans le cas concret traite ici une amelioration est permise par la stratification prealable. En outre la probabilite de circuits d'appel est tres faible, mais l'experience a montre la creation de circuits statiques dont on ne s'etait pas apercu, ne correspondant a aucun circuit des appels dynamiques quel que soit le jeu de donnees. Cela justifie a posteriori l'enumeration.

7.1.3. Enumeration des circuits.

Une des contraintes de la programmation structuree, et qui s'applique donc a l'utilisation des structures chainees, est la necessite d'algorithmes correctement formalises, utilisant au depart les seuls operateurs primitifs de la theorie mathematique sous-jacente. Cette contrainte entraine parfois la necessite de reconstruire certains algorithmes initialement fournis sous une forme inadaptée, en general trop proche des representations. C'est le cas de l'algorithme de Tiernan que nous devons donc modifier et en consequence redemontrer.

Soit donc un graphe $G=(X,)$ muni d'un ordre total sur l'ensemble des sommets et d'ordre totaux sur les ensembles de successeurs de chaque sommet, et en consequence aussi sur les ensembles d'arcs sortants. Ces ordres peuvent etre les restrictions d'un ordre sur tous les sommets mais ce n'est nullement necessaire. Ces ordres peuvent parfaitement etre lies a une methode de balayage au niveau de la representation.

Soit alors l'algorithme suivant, ou CH designe un chemin du graphe ou une suite vide:

```

: balayer les sommets par X faire
:   debut   XX := X
:   CH := le chemin reduit a X
:   Z := le premier successeur de X
:   tant que CH n'est pas vide faire
:     debut
:       { ici commence la region d'allongement du chemin }
:       boucle { structure "loop/exit" classique }
:         chercher un sommet Y qui soit a la fois
:           successeur de X,
:           plus grand que Z en tant que successeur,
:           plus grand que X en tant que sommet,
:           n'existant pas dans CH ;
:         si non trouve alors exit1
:         allonger CH en queue par Y
:         XX := Y
:         Z := le premier successeur de XX
:       fin de boucle
:       { ici commence la region d'utilisation }
:       si il y a un arc de la fin de CH vers X alors UTILISER(CH)
:       Y := sommet fin de CH
:       { ici commence la region de raccourcissement du chemin }
:       enlever le dernier sommet de CH
:       si CH non vide alors Z := le suivant de Y parmi
:         les successeurs de la fin de CH
:       fin du tant que
: fin de l'algorithme
```

Cette forme d'algorithme ne prejuge d'aucun choix de representation, la pile traditionnelle revenant a sa signification naturelle de representation d'un chemin dans le graphe. Le balayage externe sera en general dans l'ordre des sommets, mais ce n'est obligatoire ni pour le fonctionnement ni pour la demonstration.

Pour demontrer on commence par eteblir une relation d'ordre total sur les chemins issus d'un meme sommet. On dira que $C \leq D$ si, en prenant le segment initial commun maximal $C[0..i] = D[0..i]$ on a $C[i+1] \leq D[i+1]$ au sens de l'ordre sur les successeurs de C_i . La demonstration est celle de l'ordre lexicographique habituel, les hypotheses utiles etant les memes. Par contre nous prendrons ici $C \leq D$ lorsque D est un segment initial de C .

Nous appellerons chemin (resp. circuit) convenable tout chemin (resp. circuit) elementaire commençant par X , ne contenant aucun sommet inferieur a X . Compte tenu de leur construction, tous les chemins construits par l'algorithme sont convenables. De meme les chemins disponibles dans la region d'utilisation sont dans l'ordre total defini ci-dessus, et en particulier le calcul s'arrete et il n'y a pas de repetitions.

Il reste a montrer que tous les chemins convenables sont effectivement obtenus. Pour cela on considere d'abord le premier chemin disponible dans la zone d'utilisation C_1 . Pour tout autre chemin D , on montre aisement $C_1 \leq D$, donc C_1 est aussi le premier pour l'ordre. Ensuite pour deux chemins obtenus consecutivement C_i et C_{i+1} on montre de meme $D \leq C_i \Rightarrow D \leq C_{i+1}$ ce qui termine la demonstration de l'enumeration.

Ainsi l'instruction commandee par la boucle externe correspond a une enumeration des chemins convenables issus du sommet X . L'algorithme donne donc bien une enumeration des circuits elementaires. Nous obtenons finalement un algorithme raffinement en listes, alors que la version initiale en tableaux etait inutilisable directement. Soulignons au passage que l'effort de demonstration est sensiblement moindre sous la forme generale que sous la forme particuliere.

7.2. ALGORITHME EXPRIME EN STRUCTURES CHAINEES

7.2.1. Elaboration des representations.

Le choix d'une representation depend fortement de la connaissance qu'on peut avoir de structures classiques et de leur emploi, de leurs avantages en telle ou telle circonstance. Le choix des operateurs primitifs depend aussi d'idees preconçues sur le niveau suivant de raffinement. Parmi divers systemes equivalents d'operateurs primitifs, l'etude de leurs avantages respectifs prend necessairement en compte ce que l'on fera ulterieurement. Le choix des structures chainees exprime ouvertement a priori n'a pas les memes consequences que la volonte d'utiliser au maximum des tableaux, par exemple.

Nous avons dans ce projet une situation relativement classique, presque purement ensembliste, permettant le choix des structures classiques de listes lineaires lineaires ou binaires dans leurs conditions naturelles d'emploi. La situation de 1.4 est sensiblement differente, ou il est preferable au contraire d'introduire une structure originale adaptee au probleme. Ici, a partir de nos connaissances prealables, nous pourrions fixer rapidement nos choix.

Parmi les ensembles fondamentaux de notre objet externe, deux sont associes a un besoin de construction a partir des noms issus de la lecture du jeu de donnees. Nous associerons donc une structure de liste binaire ordonnee a P et E. Structurer l'ensemble X des references externes est plus complexe. Cet ensemble n'a pas d'ordre naturel puisqu'au sens strict c'est une somme d'ensemble (encore appelee "somme disjointe"). Un leger retour sur le probleme a resoudre et l'etat actuel de l'algorithme montre qu'il est inutile de structurer X lui-meme, qu'il est suffisant de structurer les ensembles PX(p).

Les autres ensembles PE(p), PX(p), EX(e) ainsi que P lui-meme sont associes dans l'algorithme a des besoins de balayage, ce qui suggere fortement l'emploi de listes lineaires. Par contre la richesse en informations de chainage (liste double ou non?) comme le caractere ordonne ou non de ces listes peuvent etre differes assez longtemps. En plus ces deux caracteres peuvent parfaitement varier sans difficulte d'une partie a l'autre du calcul, ce qui permet des choix comme les suivants:

- toutes les listes lineaires sont simples et ordonnees
- sauf indication contraire les listes ordonnees sont croissantes
- les listes de P et des PX(p) sont creees decroissantes et inversees au moment de la creation du graphe (pour des raisons d'efficacite).

Avant de definir le graphe et ses annexes notons l'interet qu'il peut y avoir a structurer l'ensemble des moules dans le but de traiter par des sous-programmes unifies successeurs et predecesseurs, et plus generalement les notions associees par changement de l'orientation du graphe. Cette possibilite est laissee ouverte aussi bien par les fondements logiques (voir CPPE section 2.2) que par le refus de

structurer les moules dans leur definition (voir 3.1.1 emploi de la notion de famille). La notion associee dans le langage est naturellement un tableau avec comme type d'indice un scalaire a deux elements.

Terminons maintenant la definition des representations:

- le graphe est represente par des listes croissantes d'arcs entrants et sortants, chaque arc referencant ses sommets extremités;
- la fermeture transitive est analogue au graphe de depart;
- les strates sont organisees en listes croissantes, leurs tetes en une liste lineaire simple;
- les chemins sont des couples formes d'une pile (cas particulier de liste) d'arcs et du sommet terminal (eventuellement non defini lorsque la pile est vide.

7.2.2. Quelques extraits de l'algorithme en listes.

Pour faciliter le suivi du texte ce n'est qu'en annexe que nous donnons une partie de l'algorithme exprime en termes de listes. Ce n'est pour l'essentiel qu'une transcription dans un formalisme different, et on montre sans difficulte que les ensembles d'elements des diverses listes ont bien les proprietes des objets initiaux.

Notons cependant la necessite de transformer "Roy-Warshall" pour le ramener a une efficacite du troisieme degre par une methode d'interclassement des arcs. Avec des listes non ordonnees l'efficacite de degre trois est aussi possible en creant une sous-construction intermediaire qui est ensuite detruite.

7.2.3. Definition de la structure chainee.

Le passage d'une structure definie en listes a une structure chainee n'est jamais totalement automatique, meme si ses grandes lignes sont deja contenues dans la version initiale. Un algorithme exprime en types abstraits ne se preoccupe pas de l'endroit (au sens le plus general) ou se trouvent les informations necessaires. Les structures chainees sont la pour introduire cette idee de place tout en demeurant dans un cadre abstrait et formalise.

Dans ce processus apparaissent des questions du genre: "faut-il conserver telle information ou bien allons-nous la recalculer au besoin?". Dans le premier cas on se demande ensuite "ou?" et dans le second "comment?". Ces choix ont eventuellement une importance tres tardive. Ce n'est par exemple que tres tardivement que l'on decidera de l'existence du chainage reciproque (liste lineaire double ou chainage physique permanent de remontee en liste binaire).

Un autre type de questions apparait frequemment. Nous utilisons un grand nombre de listes de meme nature, ce nombre dependant de la taille des donnees. Dans notre exemple de 1.4 il y a une liste d'arborescences par nombre entier. Dans un graphe il y a une liste d'arcs sortants (resp. entrants) par sommet. Il faut alors trouver une propriete permettant de reduire le nombre de listes contenant chaque sommet a une quantite statiquement bornee. Si cette propriete est inexistante ou introuvable il faut continuer a raffiner la structure. La situation la plus agreable est celle ou on peut partitionner l'ensemble des elements concernes, chacun appartenant au plus a une liste, avec une information ou un algorithme permettant de savoir a quelle liste particuliere il appartient. C'est le cas pour les graphes ou le sommet origine caracterise l'unique liste d'arcs sortants pour chaque arc. C'est le cas pour les ensembles PE(p), PX(p), EX(e) de notre projet, chaque element appartenant a l'unique liste caracterisee par la fonction reciproque EP, XP, XE.

L'exemple le plus simple ou un raffinement est necessaire est la version minimale de graphe, donnee pour chaque sommet par une liste de sommets successeurs. Chaque sommet appartient alors a un nombre imprevisible de listes, ce qui contraint a introduire une notion d'arc, telle qu'elle apparait ici. La situation peut etre tres differente pour des graphes particuliers ou on sait borner le nombre de successeurs.

Pour traiter notre exemple on peut se fixer le principe de depart du nombre minimal de moules permettant la representation. Nous obtenons ainsi la structure definie par les deux tableaux ci-contre definissant le squelette.

=====			
OBJET reel	type	informations	
=====			
sous-programme	PRO	nom:NM ; niveaux:entiers	
point d'entree	ENT	nom:NM	
reference externe	EXT	information de nom indirecte	
		via la fonction EXTENT	
arcs de graphe	ARC	extremities (voir les moules)	
strate (acces)	NIV	numero implicite par la liste	
element de circuit	CIR	arc associe (voir les moules)	
=====			

Les moules munis d'une structuration complementaire sont regroupees par une accolade, une lettre minuscule indiquant l'argument employe: i/s (inf/sup) ou d/i (direct/inverse) selon le cas.

nom	domain.	codom.	axiomatique intuitive
DICPROi } DICPROs } LISPRO	PRO PRO	PRO PRO	tetes des sous-dictionnaires "inf" et "sup" des programmes suivant (liste des programmes)
PROENT SUPREN ENTPRO	PRO ENT ENT	ENT ENT PRO	premier (liste des entrees d'un pr.) suivant (idem) prog. de definition de l'entree
PROEXT SUPREX EXTPRO	PRO EXT EXT	EXT EXT PRO	premier (liste des refer. d'un pr.) suivant (idem) prog. de definition de la refer. (reciproque liste precedente)
PREADJi } PREADJd } ADJSUIi } ADJSUI d }	PRO ARC	PRO ARC	premiers (listes arcs du graphe) suivants (idem)
PRECHMi } PRECHMd } CHMSUIi } CHMSUI d }	PRO ARC	ARC ARC	premiers (listes arcs ferm. tran.) suivants (idem)
EXTREMi } EXTREMd }	ARC	PRO	extremites (partout defini)
LISNIVI } LISNIVd }	PRO	PRO	suivant (listes par niveau)
STRATEi } STRATED }	NIV	PRO	premier (idem) la fct. reciproque de la stratif. est seulement donnee par le numero
ENTEXT SUVENEX EXTENT	ENT EXT EXT	EXT EXT ENT	premier (listes externes meme nom) suivant (idem) entree de meme nom que la ref. ext
NIVSUI	NIV	NIV	suivant (liste des strates)
DICENTI } DICENTd }	ENT	ENT	tetes des sous-dictionnaires "inf" et "sup" des entrees
TPRODIC TENTDIC TNIVLIS TPROLIS	variab. variab. variab.	PRO ENT NIV PRO	tete du dictionnaire des programmes tete du dictionnaire des entrees premier (liste des strates) premier (liste des programmes)

7.2.4. Axiomatique.

L'axiomatique doit naturellement refléter l'évolution des constructions. Nous pouvons distinguer six parties successives dans l'algorithme et sept groupes de moules, associés comme suit:

=====							
	groupe de moules						
algorithme	list	dict	arcs	nivx	circ	donn	
	de P	ionn	app.	ferm			
=====							
lecture	crea	temp					crea
calcul graphe	modi		crea				MouU
niveaux	util		util		crea		
fermeture tr.	util		util	crea			
circuits	util		util		util	temp	
impressions	util		util	util	util		util
=====							

Ce tableau est rempli en termes de quatre notions fondamentales: pendant un calcul un moule peut être en situation de création, de modification, d'utilisation, ou simplement temporaire. Ces notions correspondent aux notions du chapitre 6 sur les possibilités de traitement fin des champs d'articles. Elles entraînent pour les axiomes les conséquences suivantes:

- création => rien n'est imposé aux moules en entrant dans le calcul, à la sortie leur axiomatique est satisfaite
 - modification => une axiomatique d'entrée et une de sortie doivent être prises en compte.
 - utilisation => aussi bien les axiomes que la sous-construction sont inchangés
 - temporaire => l'axiomatique est locale, rien n'est imposé aux moules à l'extérieur, et on peut en particulier les utiliser à une autre fin.
- Le tableau ci-dessus est donc un élément constitutif de la définition des axiomes de la structure. Une telle subdivision peut parfois être nécessaire à l'intérieur même d'un algorithme.

La structure est maintenant totalement définie. La donnée de l'algorithme exprime en termes de listes, le squelette et l'axiomatique de la structure chaînée constituent une documentation fondamentale pour le programme définitif, suffisante en principe pour retrouver toutes les informations pertinentes. Pour améliorer ces informations il est en général fort utile de fournir pour chaque type l'indication des divers moyens de référencer un repère. c'est une sorte de réciproque du

tableau des moules, accompagne d'un compte de references theorique demonstrable a partir des axiomes. Nous le donnons ici dans son evolution avec la meme subdivision que pour les moules. Pour simplifier on notera ici le nombre d'elements d'un ensemble (dont le nom est en majuscules) par le nom minuscule correspondant precede de "n".

typ	algorithm	compte	moyen d'accès
PRO	lecture	1	dictionnaire : TPRODUCT/DICPROi/DICPROs
	lecture	1	liste de P : TPROLIST/LISPRO
	lecture	ne	prog. de definition d'entree : ENTPRO
	lecture	nx	prog. de definition de ref. : EXTPRO
	niveaux	0 a 2	listes de stratif. : STRATE*/LISNIV*
	graphe	d (p)+	extremites d'arcs : EXTREM*
		d (p)	
	fermeture	d (p)+u	- idem -
		d (p)	
ENT	lecture	1	dictionnaire : TENTDICT/DICENTI/DICENTS
	lecture	1	entrees d'un prog. : PROENT/SUPREN
	lecture	nex(e)	entree associee a un externe : EXTENT
EXT	lecture	1	externes d'un prog. : PROEXT/SUPREX
	lecture	1	externes de meme nom : ENTEXT/SUENEX
ARC	graphe	2	listes d'arcs sortants ou entrants
			=> ou bien PREADJ*/ADJSUI*
			=> ou bien PRECHM*/CHMSUI*
	circuits	0 a 1	element de chemin : ARCCIR
NIV	niveaux	1	liste des niveaux : TNIVLIST/NIVSUI
CIR	circuits	1	pile d'arcs de chemin : PILE/CIRPIL

Ce qui nous donne a chaque fin d'algorithme les comptes de references suivants (circuits et impressions ne changent rien):

comptes en fin d'algorithme					compte
typ					maximal
	lecture	graphe	niveaux	fermeture	pendant
					circuit
PRO	$2+nx+ne + d(p)+d(p) + 0 \text{ a } 2 + d(p)+d(p)$				-
ENT	$2+nex(e)u$	-	-	-	-
EXT	2	-	-	-	-
ARC	-	2	-	-	1
	-	-	-	2	-
NIV	-	-	1	-	-
CIR	-	-	-	-	1

Ces informations complementaires sont utiles dans deux situations: - pour assurer que tout repere reste accessible; - pour elaborer un algorithme de destruction element par element, avec ou sans utilisation de jonctions "flottantes" (au sens de 3.3.1).

7.2.5. Destruction de la structure.

Le programme principal ayant ete prevu pour traiter plusieurs donnees consecutivement, un algorithme de destruction devenait necessaire. Nous avons deja evoque (3.3) la question des destructions pour souligner leur caractere delicat. Le cas particulier traite ici illustre bien les methodes a mettre en oeuvre et l'apport de la formalisation pour obtenir un algorithme correct.

7.3. PROGRAMME, RESULTATS ET ENSEIGNEMENTS

Sans doute le programme ainsi obtenu ainsi que les resultats ne rentrent-ils pas dans le cadre precis de ce travail. C'est pourquoi nous nous contenterons d'en donner en annexe quelques extraits aussi significatifs que possible.

Ce programme nous semble de taille et de complexite suffisantes pour valider les methodes employees pour sa construction. La taille est d'environ 2000 lignes de code (voir densite d'ecriture et frequence de commentaire sur les exemples) et la complexite se mesure au nombre de moules (35 permanents) et au nombre de structures pour un meme type (7 pour PRO) sans hierarchie notable entre les structures. Il est clair que des types abstraits hierarchises sont plus aises a traiter qu'une

multi-structuration d'un meme ensemble. Bien entendu le travail de construction est bien plus important que ce qui apparait dans ce chapitre puisqu'il constitue un rapport technique de taille comparable au present memoire.

Comme en toutes choses les enseignements de cette experience comportent des aspects satisfaisants et des aspects negatifs, dont il faut tirer une tendance principale. L'aspect essentiel est sans doute que la methode issue de nos considerations theoriques conduit a un programme sur, efficace et lisible dans les memes conditions que la programmation sans pointeurs. Nous irons meme jusqu'a conjecturer que l'emploi "naturel" des pointeurs dans les situations qui les justifient tend a simplifier ce qui serait sans eux une programmation "acrobatique".

Parmi les difficultes rencontrees, la plupart ont reçu une solution sous la forme de modifications deja integrees a l'expose qui precede. La difficulte permanente qu'il n'a pas ete possible de surmonter tient a l'absence d'outillage dans les langages de programmation pour traiter les pointeurs autrement que comme des adresses, au niveau le plus bas possible. Certaines acrobaties de programmation demeurent necessaires (voir chapitre 4) dans tous les langages. Il demeure une necessite permanente de raffinement "a la main" qui constitue une source permanente d'erreurs. Il ne semble pas qu'il y ait de solution en dehors de l'introduction dans les langages de l'outillage necessaire (voir chapitre 6 pour quelques suggestions).

Nous accordons enfin dans les enseignements de l'experience une grande importance au contenu pratique de la methode. Elle n'est pas seulement une construction theorique ideale, mais aussi un outil de construction de programmes reels dans des langages insuffisamment ou pas du tout adaptes.

ANNEXE : PROGRAMME

Declarations de structures.

```

STRPRO=
  record
    GDIPRO,GINPRO:PNTPRO;
    DICPRO:ARRAY[BOOLEAN] of PNTPRO;
    PRONOM:ALPHA;
    LISPRO:PNTPRO;
    PROEXT:PNTTEXT;
    PROENT:PNTENT;
    PRECHM,
    PREADJ:ARRAY[SENS] of PNTARC;
    LISNIV:ARRAY[SENS] of PNTPRO;
    RANGSO:ARRAY[SENS] of INTEGER
  end;
  { LISTES DE GESTION }
  { DICTIONNAIRE DES PROGRAMMES }
  { NOM DU PROGRAMME }
  { LISTE LINEAIRE DES PROG. }
  { TETE LISTE EXTERNES DU PROG. }
  { TETE LISTE ENTREES DU PROG. }
  { TETES LISTES CHEMINS ADJAC. }
  { TETES LISTES ARCS ADJACENTS }
  { CHAINAGE LISTE 'STRATE' }
  { VALEUR DU RANG DU SOMMET }

STREXT=
  record
    GDIEXT,GINEXT:PNTTEXT;
    SUPREX:PNTTEXT;
    SUENEX:PNTTEXT;
    EXTPRO:PNTPRO;
    EXTENT:PNTENT
  end;
  { LISTES DE GESTION }
  { CH-GE LISTE EXT. MEME PROG. }
  { CH-GE LISTE EXT. MEME NOM }
  { PROGRAMME DE DEFINITION }
  { ENTREE DE MEME NOM }

```

STRENT=	
record	
GDIENT,GINENT:PNTENT;	{ LISTES DE GESTION }
DICENT:ARRAY[BOOLEAN] of PNTENT;	{ DICT. DES ENTREES }
ENTNOM:ALPHA;	{ NOM DE L'ENTREE }
ENTEXT:PNTENT;	{ TETE LISTE EXT DE MEME NOM }
SUPREN:PNTENT;	{ SUIV. LISTE ENTREES DU PROG. }
ENTPRO:PNTPRO	{ PROGRAMME DE DEFINITION }
end;	
STRARC=	
record	
GDIARC,GINARC:PNTARC;	{ LISTES DE GESTION }
EXTREM:ARRAY[SENS] of PNTPRO;	{ EXTREMITES DES ARCS }
CHMSUI,	{ LISTES CHEMINS ADJACENTS }
ADJSUI:ARRAY[SENS] of PNTARC	{ LISTES ARCS ADJACENTS }
end;	
STRNIV=	
record	
GDINIV,GINNIV:PNTNIV;	{ LISTES DE GESTION }
NIVSUI:PNTNIV;	{ LISTE LINEAIRE DES NIVEAUX }
STRATE:ARRAY[SENS] of PNTPRO	{ TETES LISTES DES STRATES }
end;	
STRCIR=	
record	
GINCIR:PNTCIR;	{ LISTES DE GESTION }
CIRPIL:PNTCIR;	{ LISTE <<CIRCUIT>> EN PILE }
ARCCIR:PNTARC	{ ARC DU CIRCUIT }
end;	

Gestion des articles par l'utilisateur.

```
procedure INIPRO(P:PNTPRO);
{*****}
*
*   INITIALISATION OBJET PROGRAMME
*
*
*****}
begin
  WITH P↑ do
    begin
      DICPROCFALSEJ:=NIL;
      DICPROCTRUEJ:=NIL;
      LISPRO:=NIL;
      PROEXT:=NIL;
      PROENT:=NIL;
      PREADJCDIRECTJ:=NIL;
      PREADJCINVERSEJ:=NIL;
      LISNIVCDIRECTJ:=NIL;
      LISNIVCINVERSEJ:=NIL;
      RANGSOCDIRECTJ:=0;
      RANGSOCINVERSEJ:=0
    end
  end;
end;

procedure CREERPRO(var A:PNTPRO);
{*****}
*
*   ALLOCATION UNIQUE DES OBJETS
*
*
*****}
var
  Q:PNTPRO;
begin
  if TPROLIB=NIL
  then
    begin
      NEW(A);
      INIPRO(A)
    end
  else
    begin
      A:=TPROLIB;
      TPROLIB:=A↑.GDIPRO;
    end;
  Q:=TPROOCC;
  if Q#NIL then Q↑.GINPRO:=A;
  A↑.GINPRO:=NIL;
  A↑.GDIPRO:=Q;
  TPROOCC:=A;
  A↑.PRONOM:=DEFAULT
end;
```

```

procedure DESTRPRO(var A:PNTPRO);
{*****}
*
*      LIBERATION UNIQUE DES OBJETS
*
*      *****}
var
    Q,R:PNTPRO;
begin
    INIPRO(A);
    Q:=A↑.GINPRO;
    R:=A↑.GDIPRO;
    if Q#NIL then Q↑.GDIPRO:=R
                else TPROCC:=R;
    if R#NIL then R↑.GINPRO:=Q;
    A↑.GINPRO:=NIL;
    A↑.GDIPRO:=TPROLIB;
    TPROLIB:=A;
    A:=NIL;
end;

```

Sous-programme de destruction de la structure.

```
procedure DESTRUCTIONS;
{*****
*
*   DESTRUCTION TOTALE DE LA STRUCTURE
*
*****}
const
  SD=DIRECT;

                                { SENS POUR DESTR. DES ARCS }
var
  N,NN:PNTNIV;
  P,PP:PNTPRO;
  E,EE,F:PNTENT;
  X,XX:PNTXT;
  U,UU:PNTARC;
  L:BOOLEAN;
begin
  WRITELN(' DEBUT DES DESTRUCTIONS');

                                { PRIMO : NIVEAUX }
  N:=TNIVLIST;
  TNIVLIST:=NIL;
  while N#NIL do
    begin
      NN:=N;
      N:=N↑.NIVSUI;
      DESTRNIV(NN)
    end;

                                { SECUNDO : PROGRAMMES ET ARCS }
  P:=TPROLIST;
  TPROLIST:=NIL;
  TPRODUCT:=NIL;
  while P#NIL do
    begin
      U:=P↑.PRECHMCSD;
      while U#NIL do
        begin
          UU:=U;
          U:=U↑.CHMSUICS;
          DESTRARC(UU)
        end;
      PP:=P;
      P:=P↑.LISPRO;
      DESTRPRO(PP)
    end;
```

(TERTIO : ENTREES ET EXTERNES)

```
E:=TENTDICT;
TENTDICT:=NIL;
E↑.SUPREN:=NIL;
while E#NIL do
  begin
    EE:=E;
    E:=E↑.SUPREN;
    L:=TRUE;
    repeat
      F:=EE↑.DICENT[L];
      if F#NIL then
        begin
          F↑.SUPREN:=E;
          E:=F;
        end;
      L:=¬ L
    UNTIL L;
    X:=EE↑.ENTEXT;
    while X#NIL do
      begin
        XX:=X;
        X:=X↑.SUVENEX;
        DESTREXT(XX)
      end;
    DESTRENT(EE)
  end;
PROGPRINC:=NIL;
CIRCUIT:=FALSE;
WRITELN(' DESTRUCTIONS')
end { DESTRUCTIONS } ;
```

Groupe de sous-programmes de lecture de la structure.

```
procedure LECTURE;
(* **** *)
*      LECTURE DE LA STRUCTURE D'ORIGINE      *
*      *                                       *
* **** *)
```

label 1;

```
var
  P,PP,PSUP:PNTPRO;
  E,EE,F:PNTENT;
  X,Y,YY:PNTTEXT;
  EG,S:BOOLEAN;
```

```
procedure AJOVENTREE;
(* **** *)
*      AJOUT DANS LE DICTIONNAIRE DES ENTREES - COMMUN ENTR. ET EXTER.      *
*      *                                       *
* **** *)
{~} begin
{~}   EG:=FALSE;
{~}   E:=TENTDICT;
{~}   EE:=NIL;
{~}   while (E#NIL) ^ ¬ EG do
{~}     if E↑.ENTNOM=UNI.PA
{~}       then
{~}         EG:=TRUE
{~}       else
{~}         begin
{~}           S:=(E↑.ENTNOM)UNI.PA);
{~}           EE:=E;
{~}           E:=E↑.DICENTCS]
{~}         end;
{~}   if ¬ EG then
{~}     begin
{~}       CREERENT(E);
{~}       E↑.ENTNOM:=UNI.PA;
{~}       if EE#NIL
{~}         then {AJOUT NORMAL }
{~}           EE↑.DICENTCS]:=E
{~}         else { AJOUT EN TETE }
{~}           TENTDICT:=E
{~}         end
{~}   end { AJOVENTREE } ;
```

```

procedure TRAITPROGRAMME;
{*****}
#
#   TRAITEMENT D'UN NOM DE PROGRAMME EN ENTREE   #
#
{*****}
{~} begin
{~}   EG:=FALSE;
{~}   P:=TPRODUCT;
{~}   PP:=NIL;
{~}   PSUP:=NIL;
{~}   while (P#NIL) ^ ~ EG do
{~}     if P↑.PRONOM=UNI.PA
{~}       then
{~}         EG:=TRUE
{~}       else
{~}         begin
{~}           S:=(P↑.PRONOM)UNI.PA);
{~}           PP:=P;
{~}           if S then PSUP:=P;
{~}           P:=P↑.DICPROCSJ
{~}         end;
{~}   if EG
{~}   then { DEJA EXISTANT }
{~}     begin
{~}       MESSGDEB;
{~}       WRITELN('PROGRAMME DEFINI EN DOUBLE');
{~}       MESSGFIN;
{~}       ERREUR:=TRUE
{~}     end
{~}   else
{~}     begin { PROGRAMME NOUVEAU }
{~}       CREERPRO(P);
{~}       P↑.PRONOM:=UNI.PA;
{~}       if PP#NIL
{~}       then { AJOUT NORMAL }
{~}         PP↑.DICPROCSJ:=P
{~}       else { AJOUT EN TETE }
{~}         TPRODUCT:=P;
{~}         if PSUP#NIL
{~}         then { AJOUT NORMAL }
{~}           begin
{~}             P↑.LISPRO:=PSUP↑.LISPRO;
{~}             PSUP↑.LISPRO:=P
{~}           end
{~}         else { AJOUT EN TETE }
{~}           begin
{~}             P↑.LISPRO:=TPROLIST;
{~}             TPROLIST:=P
{~}           end
{~}         end
{~}     end
end { TRAITPROGRAMME } ;

```

```

procedure TRAITENTREE;
{*****}
*
*   TRAITEMENT D'UN NOM D'ENTREE
*
*   {*****}
{~}   begin
{~}   AJOVENTREE;
{~}   if E↑.ENTPRO#NIL
{~}   then
{~}       begin
{~}       MESSGDEB;
{~}       if E↑.ENTPRO=P
{~}       then
{~}           WRITELN(' REPETITION DE L'ENTREE ',E↑.ENTNOM,
{~}               ' DANS LE MEME PROGRAMME ',P↑.PRONOM)
{~}       else
{~}           begin
{~}           WRITELN(' ENTREE DEFINIE EN DOUBLE ',E↑.ENTNOM);
{~}           ERREUR:=TRUE
{~}           end;
{~}       MESSGFIN
{~}       end
{~}   else
{~}       begin
{~}       E↑.ENTPRO:=P;
{~}
{~}
{~}           { AJOUTER DANS PE(P) }
{~}       S:=TRUE;
{~}       F:=P↑.PROENT;
{~}       EE:=NIL;
{~}       while (F#NIL) ^ S do
{~}           if F↑.ENTNOM=E↑.ENTNOM
{~}           then
{~}               S:=FALSE
{~}           else
{~}               begin
{~}               EE:=F;
{~}               F:=F↑.SUPREN
{~}               end;
{~}           if EE#NIL
{~}           then { AJOUT NORMAL }
{~}               begin
{~}               E↑.SUPREN:=EE↑.SUPREN;
{~}               EE↑.SUPREN:=E
{~}               end
{~}           else { AJOUT EN TETE }
{~}               begin
{~}               E↑.SUPREN:=P↑.PROENT;
{~}               P↑.PROENT:=E
{~}               end
{~}           end
{~}       end
end { TRAITENTREE } ;

```

```

procedure TRAITEXTERNE;
{*****}
*
*   TRAITEMENT D'UN NOM D'EXTERNE EN ENTREE
*
*
{*****}
{~}   var
{~}       NOMY,NOMC:ALPHA;
{~}   begin
{~}       AJOUEXTREE;
{~}       CREEREXT(X);
{~}       X↑.EXTENT:=E;
{~}       X↑.EXTPRO:=P;
{~}
{~}                               { AJOUTER DANS EX(E) }
{~}       S:=TRUE;
{~}       EG:=FALSE;
{~}       Y:=E↑.ENTEXT;
{~}       YY:=NIL;
{~}       while (Y#NIL) ^ S ^ ¬ EG do
{~}           if Y↑.EXTPRO↑.PRONOM=P↑.PRONOM
{~}           then
{~}               EG:=TRUE
{~}           else
{~}               begin
{~}                   S:=(Y↑.EXTPRO↑.PRONOM(P↑.PRONOM));
{~}                   if S then YY:=Y;
{~}                   Y:=Y↑.SUENEX
{~}               end;
{~}       if EG
{~}       then
{~}           begin
{~}               MESSGDEB;
{~}               WRITELN(' REPETITION DE L''EXTERNE ',E↑.ENTNOM,
{~}                   ' DANS LE PROGRAMME ',P↑.PRONOM);
{~}               MESSGFIN;
{~}               DESTREXT(X);
{~}           end
{~}       else
{~}           begin
{~}               if YY#NIL
{~}               then { AJOUT NORMAL }
{~}               begin
{~}                   X↑.SUENEX:=YY↑.SUENEX;
{~}                   YY↑.SUENEX:=X
{~}               end
{~}           else { AJOUT EN TETE }
{~}           begin
{~}               X↑.SUENEX:=E↑.ENTEXT;
{~}               E↑.ENTEXT:=X
{~}           end;

```

```

{~}
{~}                                     { AJOUTER DANS PX(P) }
{~}      S:=TRUE;
{~}      EG:=FALSE;
{~}      YY:=NIL;
{~}      Y:=P↑.PROEXT;
{~}      while (Y#NIL) ∧ S ∧ ¬ EG do
{~}          if Y↑.EXTENT↑.ENTNOM=X↑.EXTENT↑.ENTNOM
{~}          then
{~}              EG:=TRUE
{~}          else
{~}              begin
{~}                  S:=(Y↑.EXTENT↑.ENTNOM)X↑.EXTENT↑.ENTNOM);
{~}                  if S then YY:=Y;
{~}                  Y:=Y↑.SUPREX
{~}                  end;
{~}          if ¬ EG
{~}          then
{~}              if YY#NIL
{~}              then { AJOUT NORMAL }
{~}                  begin
{~}                      X↑.SUPREX:=YY↑.SUPREX;
{~}                      YY↑.SUPREX:=X
{~}                  end
{~}              else { AJOUT EN TETE }
{~}                  begin
{~}                      X↑.SUPREX:=P↑.PROEXT;
{~}                      P↑.PROEXT:=X
{~}                  end
{~}          end
end { TRAITEMENTS } ;

```

```

(*****
*
*      PROCEDURE 'LECTURE' --- DEBUT DU CORPS
*
*****)
begin { LECTURE }
ERREUR:=FALSE;
while UNI.US=NOM do
begin
    TRAITPROGRAMME;
    UNISUIV;
    if UNI.US=EGALE
    then
        UNISUIV
    else
        begin
            MESSGDEB;
            WRITELN('SYMBOLE "EGALE" MANQUANT');
            MESSGFIN
        end;
    while UNI.US=NOM do
        begin
            TRAITENTREE;
            UNISUIV
        end;
    if UNI.US>EGALE then
        begin
            MESSGDEB;
            WRITELN('FIN DE LISTE INCORRECTE');
            MESSGFIN;
            ERREUR:=TRUE;
            GOTO {ERREUR SYNTAXIQUE} 1
        end;
    if UNI.US=VIRGULE then UNISUIV;
    while UNI.US=NOM do
        begin
            TRAITEXTERNE;
            UNISUIV
        end;
    if UNI.US>VIRGULE then
        begin
            MESSGDEB;
            WRITELN('FIN DE LISTE INCORRECTE');
            MESSGFIN;
            ERREUR:=TRUE;
            GOTO {ERREUR SYNTAXIQUE} 1
        end;
    if UNI.US=PTVIRG then UNISUIV
    end;
1:
WRITELN;
WRITELN(' FIN DE LECTURE');
CARCOU:=0
end { LECTURE } ;

```

Algorithme de construction du graphe des appels.

```
procedure GRAPHE;
{*****}
*
*   CALCUL DU GRAPHE A PARTIR DE LA STRUCTURE LUE
*
*   {*****}
var
  P,Q,PL2:PNTPRO;
  X,XL2:PNTXT;
  U,V,W,WW:PNTARC;
  E:PNTENT;
  INF,L:BOOLEAN;
begin
  PL2:=NIL;
  while TPROLIST#NIL do
    begin
      P:=TPROLIST;
      TPROLIST:=P↑.LISPRO;

      { P EST ISOLE }

      XL2:=NIL;
      while P↑.PROEXT#NIL do
        begin
          X:=P↑.PROEXT;
          P↑.PROEXT:=X↑.SUPREX;

          { X EST ISOLE }

          E:=X↑.EXTENT;
          Q:=E↑.ENTPRO;
          if Q#NIL
            then { AJOUTER ARC NOUVEAU ENTRE 'P' ET 'Q' }
                  begin
                    U:=Q↑.PREADJCINVERSE;
                    if U=NIL then L:=TRUE
                      else L:=(U↑.EXTREMCINVERSE)#P;
                    { "L" SIGNIFIE "ARC NOUVEAU" }
                    if L then
                      begin
                        CREERARC(V);
                        V↑.EXTREMCDIRECT]:=Q;
                        V↑.EXTREMCINVERSE]:=P;
                        V↑.ADJSUICINVERSE]:=U;
                        Q↑.PREADJCINVERSE]:=V;
                        INF:=TRUE;
                        W:=P↑.PREADJCIDIRECT;
                        WW:=NIL;
                      end;
                    {.,}
```

```

{.}      while (W#NIL) ^ INF do
          begin
            INF:=(W↑.EXTREMEDIRECT]↑.PRONOM(G↑.PRONOM));
            if INF then WW:=W;
            W:=W↑.ADJSUICDIRECT]
            end;
          if WW#NIL
          then {AJOUT NORMAL }
              begin
                V↑.ADJSUICDIRECT]:=WW↑.ADJSUICDIRECT];
                WW↑.ADJSUICDIRECT]:=V
              end
          else { AJOUT EN TETE }
              begin
                V↑.ADJSUICDIRECT]:=P↑.PREADJCDIRECT];
                P↑.PREADJCDIRECT]:=V
              end
          end
        end { AJOUTER ARC }
      else
        WRITELN(' *** EXTERNE DE BIBLIOTHEQUE '
                  ,E↑.ENTNOM,' DANS ',P↑.PRONOM);
                                     { ON REINSERE 'X' }

        X↑.SUPREX:=XL2;
        XL2:=X
        end;
        P↑.PROEXT:=XL2;
                                     { ON REINSERE 'P' }

        P↑.LISPRO:=PL2;
        PL2:=P
        end;
        TPROLIST:=PL2;
                                     { RECOPIE PREALABLE }

        P:=TPROLIST;
        while P#NIL do
          begin
            U:=P↑.PREADJCDIRECT];
            P↑.PRECHMEDIRECT]:=P↑.PREADJCDIRECT];
            P↑.PRECHMCINVERSE]:=P↑.PREADJCDIRECT];
            while U#NIL do
              begin
                U↑.CHMSUICDIRECT]:=U↑.ADJSUICDIRECT];
                U↑.CHMSUICINVERSE]:=U↑.ADJSUICINVERSE];
                U:=U↑.ADJSUICDIRECT]
              end;
            P:=P↑.LISPRO
          end;
        WRITELN(' FIN DU CALCUL DU GRAPHE')
      end { GRAPHE } ;

```

Algorithme de stratification

```
procedure NIVEAUX(SE:SENS);
{*****}
#          STRATIFICATION DU GRAPHE          #
*          #                                  #
*          #                                  #
*****}
var
  NBRPRO,RR,CPTPRO:INTEGER;
  ISE:SENS;
  T:BOOLEAN;
  P,Q,DS,SOU:PNTPRO;
  U:PNTARC;
  N,NC,DNIV:PNTNIV;
begin
  CPTPRO:=0;
  RR:=0;
  DNIV:=NIL;
  NC:=TNIVLIST;
  ISE:=OPPOSE(SE);
  P:=TPROLIST;
  while P#NIL do
    begin
      P↑.RANGSO[ISE]:=NBRPRO;
      P:=P↑.LISPRO
    end;
  repeat
    SOU:=NIL;
    DS:=NIL;
    P:=TPROLIST;
    while P#NIL do
      begin
        if P↑.RANGSO[ISE]=NBRPRO then
          begin
            T:=FALSE;
            U:=P↑.PREADJ[ISE];
            while (U#NIL) ∧ ¬ T do
              begin
                Q:=U↑.EXTREME[ISE];
                if Q↑.RANGSO[ISE]=RR
                then
                  T:=TRUE
                else
                  U:=U↑.ADJSUI[ISE]
                end;
            end;
```

{ TETE LISTE STRATE EN CONSTR. }

{ DERN. ELEM. STRATE EN CONSTR. }

```

if  $\neg$  T then
begin
if DS#NIL
then { AJOUT NORMAL }
DS↑.LISNIVEJ:=P
else { AJOUT EN TETE }
SOU:=P;
DS:=P;
CPTPRO:=CPTPRO+1;
P↑.RANGSOJSEJ:=RR
end
end;
P:=P↑.LISPRO
end;
if SOU#NIL then
begin
if NC#NIL then
begin
CREERNIV(NC);
if DNIV#NIL
then { AJOUT NORMAL }
DNIV↑.NIVSUI:=NC
else { AJOUT EN TETE }
TNIVLIST:=NC
end;
NC↑.STRATEJSEJ:=SOU;
DNIV:=NC;
NC:=NC↑.NIVSUI;
RR:=RR+1
end
UNTIL SOU=NIL;
CIRCUIT:=(CPTPRO#NBRPRO);
if SE#INVERSE then
if TNIVLIST#NIL then
begin
P:=TNIVLIST↑.STRATEJDIRECTJ;
if P#NIL then
if P↑.LISNIVJDIRECTJ=NIL then
PROGPRINC:=P
end;
if CIRCUIT then WRITELN(' IL Y A DES CIRCUITS');
if PROGPRINC=NIL then WRITELN(' ON N'A PAS DE PROGRAMME PRINCIPAL')
else WRITELN(' NOM DU PROGRAMME PRINCIPAL : ',
PROGPRINC↑.PRONOM);
WRITELN(' NOMBRE DES NIVEAUX ',CPTOCCNIVJ:5);
WRITELN(' FIN CALCUL NIVEAUX')
end { NIVEAUX } ;

```

Algorithme de recherche des circuits dans le graphe des appels.

N.B. La recherche exhaustive est cohérente dans la mesure où la probabilité de rencontre d'un circuit est faible dans le cas considéré.

```
procedure TIERNAN;
{*****}
*
*   RECHERCHE DES CIRCUITS - ALGORITHME DE TIERNAN
*
*
{*****}
const
  SD=DIRECT;
  SI=INVERSE;
  L=14;
var
  NBRPRO,K:INTEGER;
  Q,V,FIN,P:PNTPRO;
  U,ADR:PNTARC;
  INF,TR2,TR1:BOOLEAN;
  NP:ALPHA;
  A,B,PILE:PNTCIR;

begin
  WRITELN(' DEBUT DE LA RECHERCHE DES CIRCUITS');

  PILE:=NIL;
  NBRPRO:=CPTOCCCPRO;
  P:=TPROLIST;
  while P#NIL do
    begin
      if (P↑.RANGSOC[DIRECT])=NBRPRO then
      if (P↑.RANGSOC[INVERSE])=NBRPRO then
        { BALAYAGE SOMMETS SANS RANG }
        { VARIABLE CONTROLE 'P' }

        begin
          NP:=P↑.PRONOM;
          FIN:=P;
          ADR:=FIN↑.PREADJ[SD];
          { FIN EST ORIGINE DE ADR }
          { ET DESTINATION DE PILE↑.ARCCIR }

          while FIN#NIL do
            { C'A'D CHEMIN NON VIDE }

            begin
              repeat
                TR1:=FALSE;
                { TROUVER UN ARC CONVENABLE }
```

```

while (ADR#NIL) ^ ¬ TR1 do
  begin
    Y:=ADR↑.EXTREMCSDJ;
    { N.B.: ADR = ( FIN , Y ) }
    if Y↑.PRONOM(NP) then
    if Y↑.RANGSODIRECTJ)=NBRPRO then
    if Y↑.RANGSODINVERSEJ)=NBRPRO then
      begin
        { ADR CONVIENT SAUF SI EN PILE }
        A:=PILE;
        TR2:=FALSE;
        while (A#NIL) ^ ¬ TR2 do
          if A↑.ARCCIR↑.EXTREMCSDJ=ADR↑.EXTREMCSDJ
          then
            TR2:=TRUE
          else
            A:=A↑.CIRPIL;
        if ¬ TR2 then { ADR CONVIENT = } ALLONGER }
        begin
          CREERCIR(B);
          B↑.ARCCIR:=ADR;
          B↑.CIRPIL:=PILE;
          PILE:=B;
          FIN:=Y;
          ADR:=FIN↑.PREADJCSOJ;
          TR1:=TRUE
        end
      end;
    if ¬ TR1 then ADR:=ADR↑.ADJSUICSODJ
    end
  UNTIL ¬ TR1;
  { CHERCHER EXISTENCE ARC (FIN,P) }
  U:=FIN↑.PREADJCSODJ;
  TR1:=FALSE;
  INF:=TRUE;
  while (U#NIL) ^ INF ^ ¬ TR1 do
    begin
      Q:=U↑.EXTREMCSDJ;
      if Q=P
      then
        TR1:=TRUE
      else
        if Q↑.PRONOM(NP)
        then
          U:=U↑.ADJSUICSODJ
        else
          INF:=FALSE
        end;
    end;

```

```

if TR1 then
begin
WRITE(' CIRCUIT ELEMENTAIRE :      ',NP);
K:=4;
A:=NIL;
while PILE#NIL do
begin
B:=PILE;
PILE:=B↑.CIRPIL;
B↑.CIRPIL:=A;
A:=B
end;
while A#NIL do
begin
B:=A;
A:=B↑.CIRPIL;
if K)=L then begin WRITELN; K:=0 end;
WRITE(' ',B↑.ARCCIR↑.EXTREMCSD↑.PRONOM);
K:=K+1;
B↑.CIRPIL:=PILE;
PILE:=B
end;
WRITELN
end;

{ RACCOURCIR LE CHEMIN }

if PILE#NIL
then
begin
FIN:=PILE↑.ARCCIR↑.EXTREMCSD;
B:=PILE;
PILE:=B↑.CIRPIL;
ADR:=B↑.ARCCIR↑.ADJSUIESD;
DESTRCIR(B)
end
else
begin
ADR:=NIL;
FIN:=NIL
end;
end

{ SORTIE BAL. SOM. SANS RANG }

end;
P:=Pt.LISPRO
end;
WRITELN(' FIN RECHERCHE CIRCUITS')
end;

```

Fermeture transitive.

```
procedure ROYWARSHALL;
{*****}
*
*          CALCUL DE LA FERMETURE TRANSITIVE
*
*
*****}
const
  SD=DIRECT;
  SI=INVERSE;
  { OU L'INVERSE SI VOUS VOULEZ }
var
  P,Q,R,OVP,OVG:PNTPRO;
  PV,VP,VQ,U,W,PROV:PNTARC;
  INF:BOOLEAN;
  NBRARC:INTEGER;

  procedure AJOUTER;
  { UTILITAIRE LOCAL }

  begin
    R:=VP↑.EXTREMESD;
    if R=Q
    then
      WRITELN(' FERMETURE CIRCUIT AUTOUR DE ',R↑.PRONOM)
    else
      begin
        CREERARC(W);
        if PV#NIL
        then
          begin
            W↑.CHMSUICSID:=PV↑.CHMSUICSID;
            PV↑.CHMSUICSID:=W
          end
        else
          begin
            W↑.CHMSUICSID:=Q↑.PRECHMESID;
            Q↑.PRECHMESID:=W
          end;
          PV:=W;
          W↑.EXTREMESD:=Q;
          W↑.EXTREMESID:=R;
          W↑.CHMSUICSID:=PROV;
          PROV:=W
        end
      end { AJOUTER } ;
```

procedure INSERER;

{ UTILITAIRE LOCAL }

begin

while PROV#NIL do

begin

W:=PROV;

PROV:=PROV↑.CHMSUI[SD];

R:=W↑.EXTREME[SI];

Q:=W↑.EXTREME[SD];

VP:=R↑.PRECHME[SD];

PV:=NIL;

INF:=TRUE;

while (VP#NIL) ∧ INF do

begin

if VP↑.EXTREME[SD]↑.PRONOM)W↑.EXTREME[SD]↑.PRONOM

then

INF:=FALSE

else

begin

PV:=VP;

VP:=VP↑.CHMSUI[SD]

end;

end;

if PV#NIL

then

begin

W↑.CHMSUI[SD]:=PV↑.CHMSUI[SD];

PV↑.CHMSUI[SD]:=W

end

else

begin

W↑.CHMSUI[SD]:=R↑.PRECHME[SD];

R↑.PRECHME[SD]:=W

end

end;

end { INSERER } ;

```

begin
  {*****}
  *
  *   CORPS DE 'ROY-WARSHALL'   *
  *
  {*****}

  WRITELN(' FERMETURE TRANSITIVE');

  NBRARC:=CPTOCCEARC];

                                     { CALCUL PRINCIPAL }

  P:=TPROLIST;
  while P#NIL do
    begin
      PROV:=NIL;
      U:=P↑.PRECHMCSD];
      while U#NIL do
        begin
          Q:=U↑.EXTREMCSD];

                                     { AJOUTER PREDCP A PREDCQ }

          PV:=NIL;
          VQ:=Q↑.PRECHMCSD];
          VP:=P↑.PRECHMCSD];
          while (VP#NIL) ^ (VQ#NIL) do
            begin
              OVP:=VP↑.EXTREMCSD];
              OVQ:=VQ↑.EXTREMCSD];
              if OVP=OVQ then

                                     { ARC ANCIEN ET NOUVEAU }

                begin
                  PV:=VQ;
                  VQ:=VQ↑.CHMSUI[SI];
                  VP:=VP↑.CHMSUI[SI];
                end
              else if OVP↑.PRONOM)OVQ↑.PRONOM then
                                     { ARC SEULEMENT NOUVEAU }
                begin
                  PV:=VQ;
                  VQ:=VQ↑.CHMSUI[SI];
                end
              else

                                     { ARC SEULEMENT ANCIEN }

                begin
                  AJOUTER;
                  VP:=VP↑.CHMSUI[SI];
                end
            end;
          end;
        end;
      end;
    end;
  end;

```

```

while VP#NIL do
    begin
        AJOUTER;
        VP:=VP↑.CHMSUI[SI]
    end;
while VP#NIL do
    begin
        AJOUTER;
        VP:=VP↑.CHMSUI[SI]
    end;
    U:=U↑.CHMSUI[SD]
end;
INSERER;
P:=P↑.LISPRO
end;
WRITELN(' NOMBRE DE NOUVEAUX ARCS ',CPTOCC[ARC]-NBRARC);
WRITELN(' FIN FERMETURE TRANSITIVE')
end { ROY-WARSHALL } ;

```


BIBLIOGRAPHIE

- [ABR77] ABRIAL J.R.
Le langage "Z"
Groupe Programmation et Langages de l'AFCEt - juin 1977
Bulletin de l'A.F.C.E.T no 3
- [ARS78] ARSAC J.
An interactive program manipulation system for non-naive users.
Publication du L.I.T.P no 78-10
- [BOH66] BOHM C. et JACOPINI G.
Flow diagrams, Turing machines and languages with only
two formation rules.
C.A.C.M. 9/66
- [BOU58] BOURBAKI N.
Elements de mathematiques - livre 1
fascicule de resultats
Hermann - 1958
- [BUR76] BURSTALL et DARLINGTON
A system which automatically improves programs.
Acta Informatica 6/76
- [COR78] CORRELL C. H.
Proving programs correct through refinement.
Acta Informatica 9/78
- [COU77] COUSINEAU G.
Les arbres a feuilles indicees: un cadre algebrique pour
l'etude des structures de controle.
These d'etat - Univ. Pierre et Marie Curie - mai 1977
- [DAH72] DAHL C. A. R, DIJKSTRA E. W. et HOARE C. A. R
Structured programming.
Academic press - 1972

- [DEM77] DEMBINSKI et SCHWARTZ
The taming of the pointer.
Sigplan 7/77
- [FLO67] FLOYD R.
Assigning meaning to programs.
Symposium on applied mathematics - Providence Rhode Island
American Mathematical Society 1967
- [GAU74] GAUTHIER M. P.
Le P-Code de Pascal.
Rapport technique Institut de Programmation - juin 1974
- [GUT75] GUTTA J. V.
The specification and application to programming of
abstract data types.
Ph.D - Univ. de Toronto - septembre 1975
- [HOA69] HOARE C. A. R.
An axiomatic basis for computer programming.
C.A.C.M. 12/69
- [HOA73] HOARE C. A. R. et WIRTH N.
Axiomatic definition of the programming language Pascal.
Eidgenossische Technische Hochschule Zurich - 1973
- [KRE67] KREISEL G. et KRIVINE J. L.
Elements de logique mathematique - theorie des modeles.
Dunod (Paris) - 1967
- [KRI69] KRIVINE J. L.
Theorie axiomatique des ensembles.
Presses Universitaires de France (Paris) - 1969
- [LAU76] LAURIERE J. L.
Un langage et un programme pour enoncer et resoudre des
problemes combinatoires.
These d'etat - Univ. Pierre et Marie Curie - 1976

- [LIS75] LISKOV B. et ZILLES S.
Specification techniques for data abstraction.
IEEE transactions on software engeneering -
vol SE-1 - 1975
- [LIS78] LISKOV B. et JONES A. K.
A language extension for expressing constraints on
data access.
C.A.C.M. 5/78
- [NGU78] NGUYEN Van Lu
Contribution a l'etude des types polymorphes.
These 3e cycle - Univ. Pierre et Marie Curie - 1978
- [SC071] SCOTT D. et STRACHEY C.
Towards a mathematical semantics for computing languages.
Proc. symposium on computers and automata
Polytechnic Institute of Brooklyn Press (N.Y.) - 1971
- [TIE70] TIERNAN J. C.
An efficient search algorithm to find the elementary
circuits of a graph.
C.A.C.M. 12/70
- [WAR77] WARREN D.H.D., PEREIRA L. M. et PEREIRA F.
"Prolog", the language and its implementation, ...
A.C.M. symposium on Artif. Intel. and Programming Languages.
Rochester - 1977
- [WIR73] WIRTH N.
The Programming Language Pascal
Eidgenossische Technische Hochschule Zurich - 1973
- [WIR76] WIRTH N.
Algorithms + data structures = programs.
Prentice hall - 1976
- [WUL75] WULF W. A., LONDON R. L. et SHAW M.
Abstraction and verification in Alphard.
New directions in algorithmic languages (ed. S.A. Schuman)
I.R.I.A. 1975

